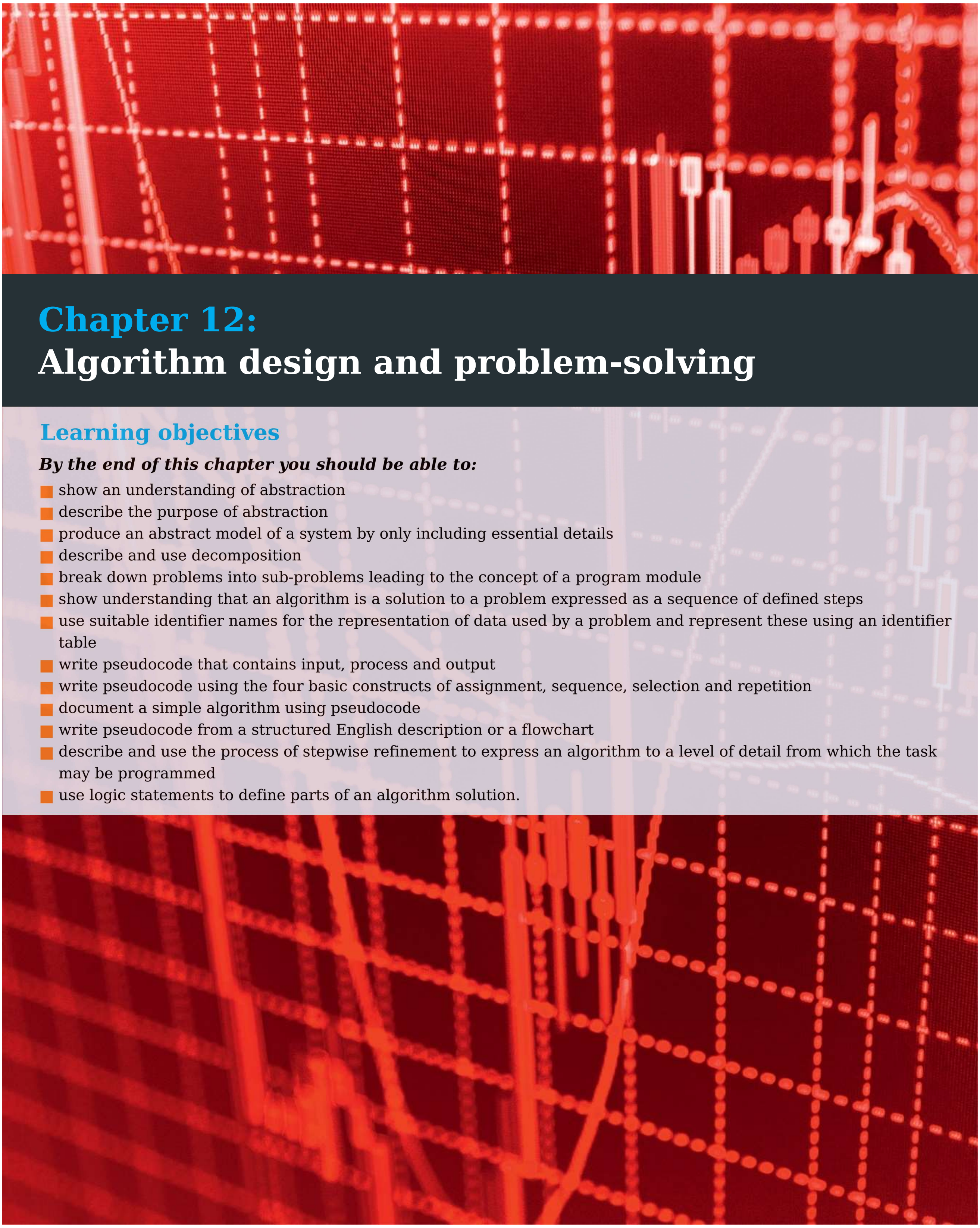


Chapter 12:
Algorithm design and problem-solving

Learning objectives

By the end of this chapter you should be able to:

- show an understanding of abstraction
- describe the purpose of abstraction
- produce an abstract model of a system by only including essential details
- describe and use decomposition
- break down problems into sub-problems leading to the concept of a program module
- show understanding that an algorithm is a solution to a problem expressed as a sequence of defined steps
- use suitable identifier names for the representation of data used by a problem and represent these using an identifier table
- write pseudocode that contains input, process and output
- write pseudocode using the four basic constructs of assignment, sequence, selection and repetition
- document a simple algorithm using pseudocode
- write pseudocode from a structured English description or a flowchart
- describe and use the process of stepwise refinement to express an algorithm to a level of detail from which the task may be programmed
- use logic statements to define parts of an algorithm solution.

12.01 What is computational thinking?

Computational thinking is a problem-solving process where a number of steps are taken in order to reach a solution. It is a logical approach to analysing a problem, producing a solution that can be understood by humans and used by computers.

Computational thinking involves five key strands: abstraction, decomposition, data modelling, pattern recognition and algorithmic thinking.

Abstraction

Abstraction involves filtering out information that is not necessary to solve a problem. There are many examples in everyday life where abstraction is used. Figure 12.02 shows part of the underground map of London, UK. The purpose of this map is to help people plan their journey within London. The map does not show a geographical representation of the tracks of the underground train network nor does it show the streets above ground. It shows the stations and which train lines connect the stations. In other words, the information that is not necessary when planning how to get from one landmark to another is filtered out. The essential information we need to be able to plan our route is clearly represented.

Abstraction gives us the power to deal with complexity. An algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs. An abstract data type defines an abstract set of values and operations for manipulating those values.

TASK 12.01

Use the aerial photograph in Figure 12.01 and draw a map just showing the essential details for finding a route from landmark A to landmark B.

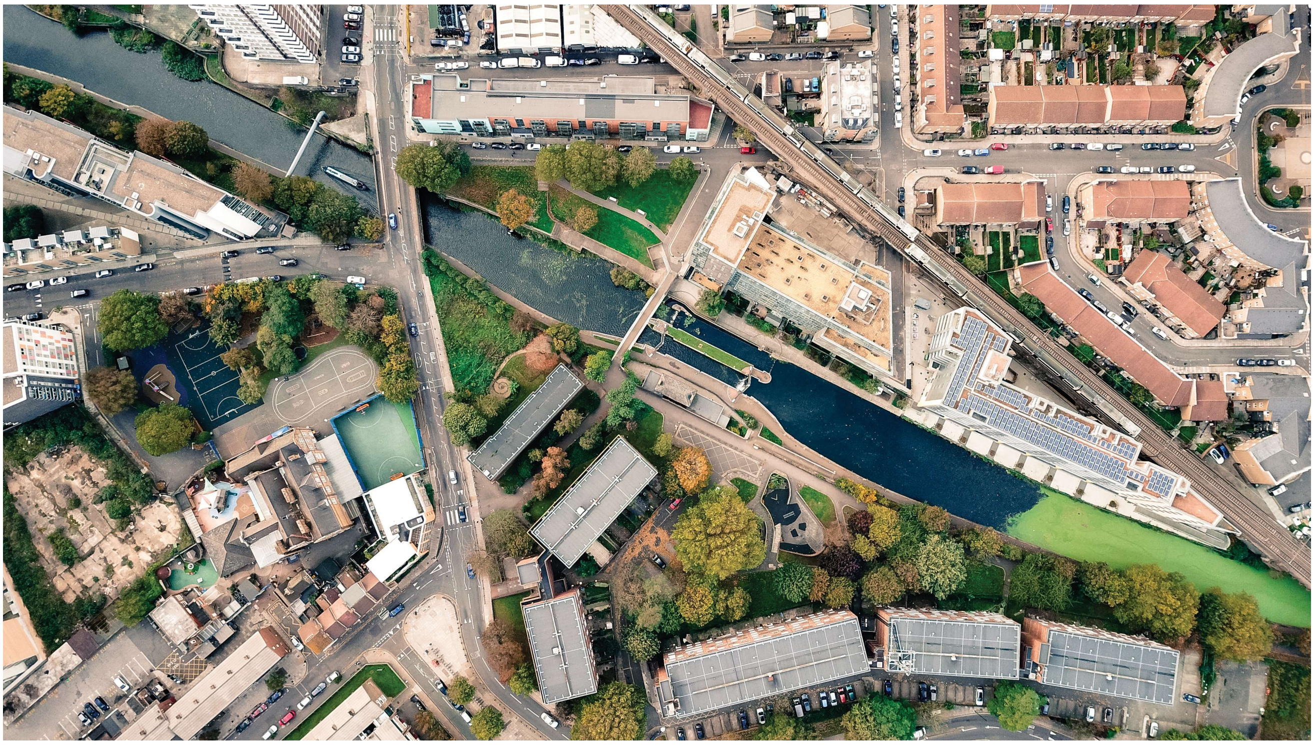


Figure 12.01 Aerial photograph of part of a city

Decomposition

Decomposition means breaking problems down into sub-problems in order to explain a process more clearly. Decomposition leads us to the concept of program modules and using procedures and functions.

Data modelling

Data modelling involves analysing and organising data (see Chapter 13). We can set up abstract data types to model real-world concepts, such as queues or stacks. When a programming language does not have such data types built-in, we can define our own by building them from existing data types. There are more ways to build data models. In Chapter 27 we cover object-oriented programming where we build data models by defining classes. In Chapter 29 we model data using facts and rules. In Chapter 26 we cover random files.

Pattern recognition

Pattern recognition means looking for patterns or common solutions to common problems and using these to complete tasks in a more efficient and effective way. There are many standard algorithms to solve standard problems, such as sorting (see Section 13.03 and 23.03) or searching (see Section 13.02 and 23.04).

Algorithm design

Algorithm design involves developing step-by-step instructions to solve a problem.

12.02 What is an algorithm?

We use algorithms in everyday life. If you need to change a wheel on a car, you might need to follow instructions (the algorithm) from a manual.

1. Take a spanner and loosen the wheel nuts.
2. Position a jack in an appropriate place.
3. Raise the car.
4. Take off the wheel nuts and the wheel.
5. Lift replacement wheel into position.
6. Replace wheel nuts and tighten by hand.
7. Lower the car.
8. Fully tighten wheel nuts.

This might sound all very straightforward. However, if the instructions are not followed in the correct logical sequence, the process might become much more difficult or even impossible. For example, if you tried to do Step 1 after Step 3, the wheel may spin and you can't loosen the wheel nuts. You can't do Step 4 before Step 3.

If you want to bake a cake, you follow a recipe.

1. Measure the following ingredients: 200g sugar, 200g butter, 4 eggs, 200g flour, 2 teaspoons baking powder and 2 tablespoons of milk.
2. Mix the ingredients together in a large bowl, until the consistency of the mixture is smooth.
3. Pour the mixture into a cake tin.
4. Bake in the oven at 190° C for 20 minutes.
5. Check it is fully cooked.
6. Turn cake out of the tin and cool on a wire rack.

The recipe is an algorithm. The ingredients are the input and the cake is the output. The process is mixing the ingredients and cooking the mixture in the oven.

Sometimes a step might need breaking down into smaller steps. For example, Step 2 can be more detailed.

2.1 Beat the sugar and butter together until fluffy.

2.2 Add the eggs, one at a time, mixing constantly.

2.3 Sieve the flour and baking powder and stir slowly into the egg mixture.

2.4 Add milk and mix to give a creamy consistency.

Sometimes there might be different steps depending on some other conditions. For example, consider how to get from one place to another using the map of the London Underground system in Figure 12.02.

images

Figure 12.02 Underground map of London, UK

To travel from King’s Cross St. Pancras to Westminster, we consider two routes:

- Route A: Take the Victoria Line to Green Park (4 stations); then take the Jubilee Line to Westminster (1 station)
- Route B: Take the Piccadilly Line to Green Park (6 stations); then take the Jubilee Line to Westminster (1 station).

Route A looks like the best route. If there are engineering works on the Victoria Line and trains are delayed, Route B might turn out to be the quicker route.

The directions on how to get from King’s Cross St. Pancras to Westminster can be written as:

```
IF there are engineering works on the Victoria Line
   THEN
     Take the Piccadilly Line to Green Park (6 stations)
     Take the Jubilee Line to Westminster (1 station)
   ELSE
     Take the Victoria Line to Green Park (4 stations)
     Take the Jubilee Line to Westminster (1 station)
```

TASK 12.02

Write the steps to be followed to:

- make a sandwich
- walk from your school/college to the nearest shop
- log on to your computer.

Many problems have more than one solution. Sometimes it is a personal preference which solution to choose. Sometimes one solution will be measurably better than another.

12.03 Expressing algorithms

TIP

Computer scientists are interested in finding good solutions. A good solution gives the correct results, takes up as little computer memory as possible and executes as fast as possible. The solution should be concise, elegant and easy to understand.

In computer science, when we design a solution to a problem we express the solution (the algorithm) using sequences of steps written in **structured English** or **pseudocode**. Structured English is a subset of the English language and consists of command statements. Pseudocode resembles a programming language without following the syntax of a particular programming language. A **flowchart** is an alternative method of representing an algorithm. A flowchart consists of specific shapes, linked together.

An algorithm consists of a sequence of steps. Under certain conditions we may wish not to perform some steps. We may wish to repeat a number of steps. In computer science, when writing algorithms, we use four basic types of construct.

- **Assignment:** a value is given a name (identifier) or the value associated with a given identifier is changed.
- **Sequence:** a number of steps are performed, one after the other.
- **Selection:** under certain conditions some steps are performed, otherwise different (or no) steps are performed.
- **Repetition:** a sequence of steps is performed a number of times. This is also known as iteration or looping.

Many problems we try to solve with a computer involve data. The solution involves inputting data to the computer, processing the data and outputting results (as shown in Figure 12.03).

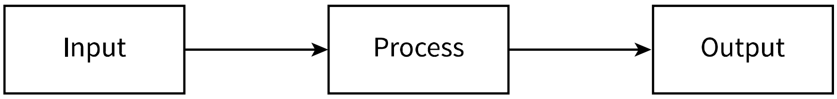
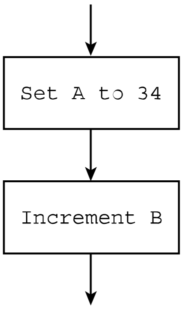


Figure 12.03 Input–process–output

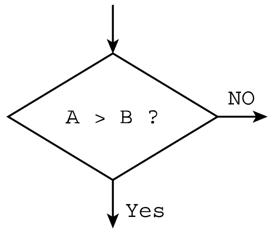
We therefore also need input and output statements.

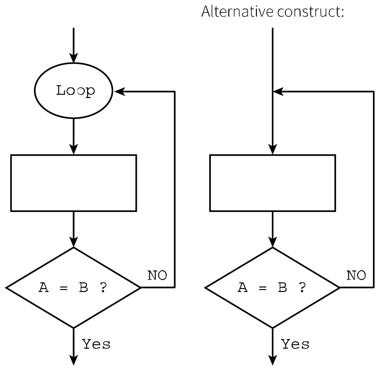
We need to know the constructs so we know how detailed our design has to be. These constructs are represented in each of the three notations as shown in Table 12.01.

In this book, algorithms and program code are typed using the `Courier` font.

	Structured English	Pseudocode	Flowchart
Assignment and Sequence	`SET A TO 34` `INCREMENT B`	`A ← 34` `B ← B + 1`	

Selection	`IF A IS GREATER THAN B`
	`     THEN ...`
	`     ELSE ...`
		`IF A > B`
		`   THEN ...`
		`   ELSE ...`
		`ENDIF`	

Repetition	`REPEAT UNTIL A IS EQUAL TO B ...`	`REPEAT`
		`...`
		`UNTIL A = B`
			 Alternative construct:
			Loop
			A = B ?
			NO
			Yes
			A = B ?
			NO
			Yes

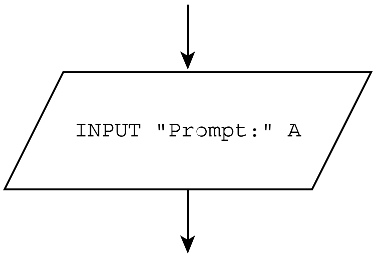
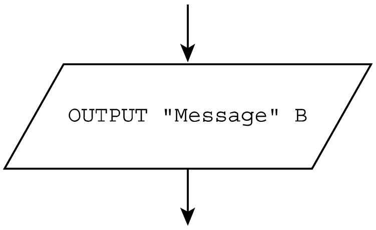
Input	`INPUT A`	`INPUT "Prompt: " A`	 INPUT "Prompt:" A
Output	`OUTPUT "Message"` `OUTPUT B`	`OUTPUT "Message", B`	 OUTPUT "Message" B

Table 12.01 Constructs for computing algorithms

12.04 Variables

When we input data for a process, individual values need to be stored in memory. We need to be able to refer to a specific memory location so that we can write statements of what to do with the value stored there. We refer to these named memory locations as **variables**. You can imagine these variables like boxes with name labels on them. When a value is input, it is stored in the box with the specified name (identifier) on it.

For example, the variable used to store a count of how many guesses have been made in a number guessing game might be given the identifier `NumberOfGuesses` and the player's name might be stored in a variable called `ThisPlayer`, as shown in Figure 12.04.

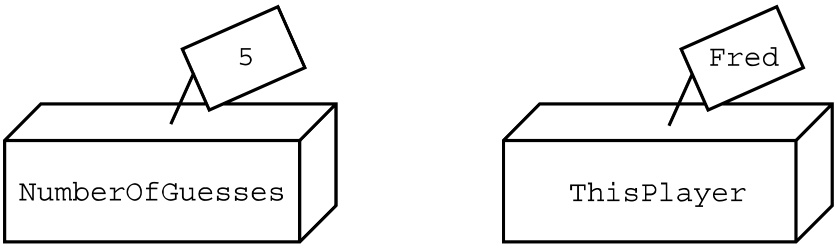


Figure 12.04 Variables

Variable identifiers should not contain spaces, only letters, digits and _ (the underscore symbol). To make algorithms easier to understand, the naming of a variable should reflect the variable's use. This means often that more than one word is used as an identifier. The formatting convention used here is known as CamelCaps. It makes an identifier easier to read.

12.05 Assignments

Assigning a value

The following pseudocode stores the value that is input (for example 15) in a variable with the identifier `Number` (see Figure 12.05(a)).

```
INPUT Number
```

The following pseudocode stores the value 1 in the variable with the identifier `NumberOfGuesses` (see Figure 12.05(b)).

```
NumberOfGuesses ← 1
```

15
Number
a
1
NumberOfGuesses
b

Figure 12.05 Variables being assigned a value

Updating a value

The following pseudocode takes the value stored in `NumberOfGuesses` (see Figure 12.06 (a)), adds 1 to that value and then stores the new value back into the variable `NumberOfGuesses` (see Figure 12.06 (b)).

```
NumberOfGuesses ← NumberOfGuesses + 1
```

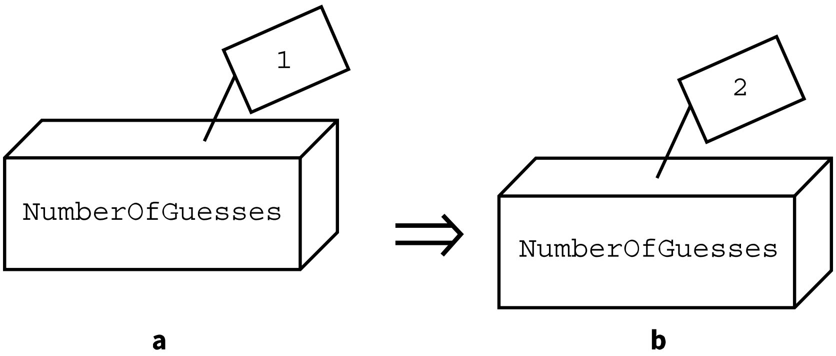


Figure 12.06 Updating the value of a variable

Copying a value

Values can be copied from one variable to another.

The following pseudocode takes the value stored in `Value1` and copies it to `Value2` (see Figure 12.07).

```
    Value2 ← Value1
```

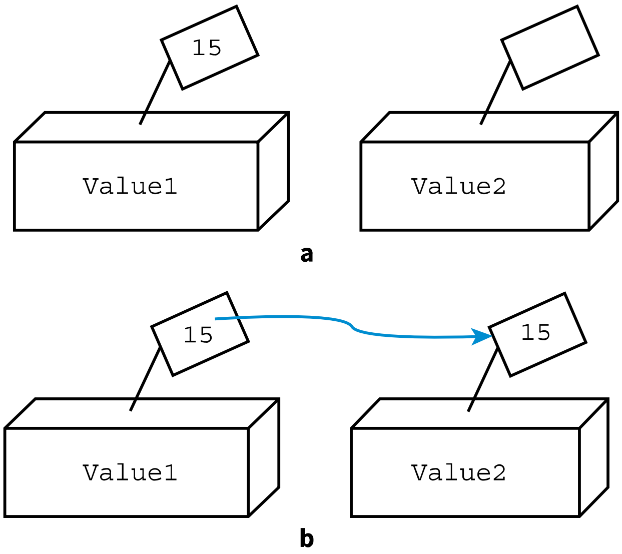

15
Value1
Value2
a
15
15
Value1
Value2
b

Figure 12.07 Copying the value of a variable

The value in `Value1` remains the same until it is assigned a different value.

Swapping two values

If we want to swap the contents of two variables, we need to store one of the values in another variable temporarily. Otherwise the second value to be moved will be overwritten by the first value to be moved.

In Figure 12.08(a), we copy the content from `Value1` into a temporary variable called `Temp`. Then we copy the content from `Value2` into `Value1` Figure 12.08(b)). Finally, we can copy the value from `Temp` into `Value2` (Figure 12.08(c)).

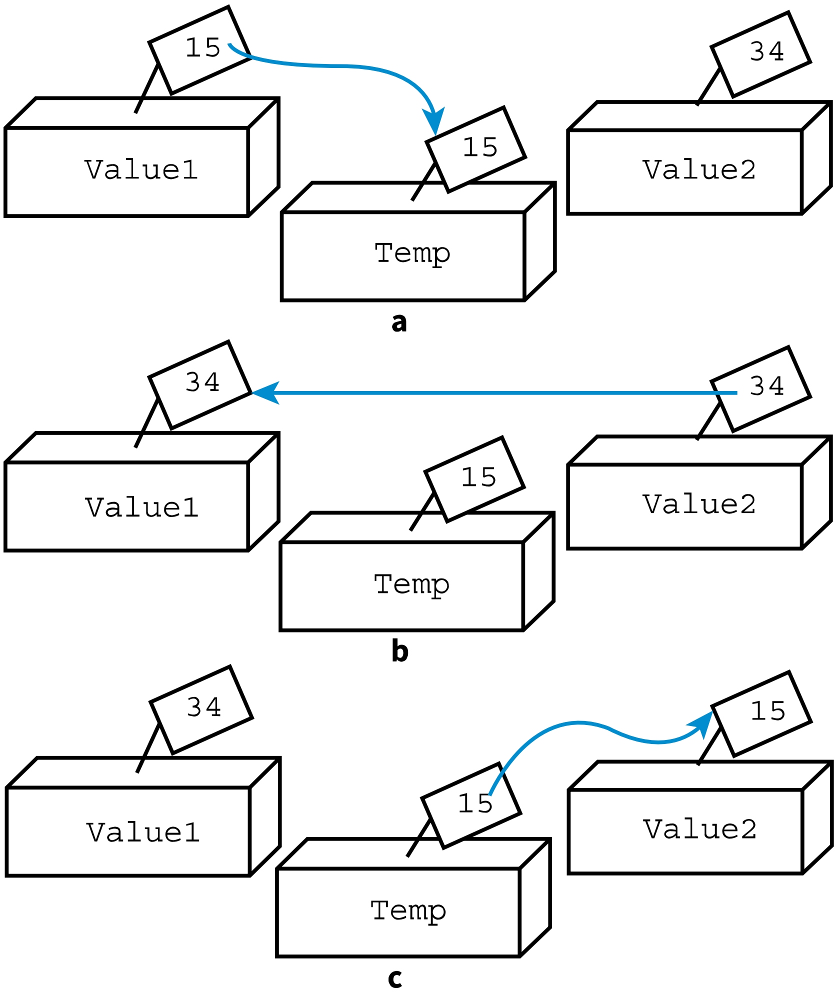

15
Value1
15
Temp
34
Value2
a
34
Value1
15
Temp
34
Value2
b
34
Value1
15
Temp
15
Value2
c

Figure 12.08 Swapping the values of two variables

Using pseudocode we write:

```
Temp ← Value1
Value1 ← Value2
Value2 ← Temp
```

WORKED EXAMPLE 12.01

Using input, output, assignment and sequence constructs

The problem to be solved: Convert a distance in miles and output the equivalent distance in km.

Step 1: Write the problem as a series of structured English statements:

```
INPUT number of miles
Calculate number of km
OUTPUT calculated result as km
```

Step 2: Analyse the data values that are needed.

We need a variable to store the original distance in miles and a variable to store the result of multiplying the number of miles by 1.61. It is helpful to construct an **identifier table** to list the variables.

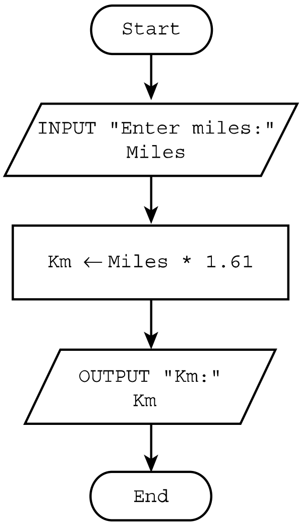
Identifier	Explanation
`Miles`	Distance as a whole number of miles
`Km`	The result from using the given formula: Km = Miles * 1.61

Table 12.02 Identifier table for miles to km conversion

Step 3: Provide more detail by drawing a flowchart or writing pseudocode.

The detail given in a flowchart should be the same as the detail given in pseudocode. It should use the basic constructs listed in Table 12.01.

Figure 12.09 represents our algorithm using a flowchart and the equivalent pseudocode.



```
INPUT "Enter miles:" Miles
Km ← Miles * 1.61
OUTPUT "km:", Km
```

Figure 12.09 Flowchart and pseudocode for miles to km conversion

TASK 12.03

Consider the following algorithm steps.

1 Input a length in inches.

2 Calculate the equivalent in centimetres.

3 Output the result.

List the variables required in an identifier table.

Write pseudocode for the algorithm.

12.06 Logic statements

In Section 12.02, we looked at an algorithm with different steps depending on some other condition:

```
IF there are engineering works on the Victoria Line
   THEN
     Take the Piccadilly Line to Green Park (6 stations)
     Take the Jubilee Line to Westminster (1 station)
   ELSE
     Take the Victoria Line to Green Park (4 stations)
     Take the Jubilee Line to Westminster (1 station)
```

The selection construct in Table 12.01 uses a condition to follow either the first group of steps or the second group of steps (see Figure 12.10).

A condition consists of at least one logic proposition (see Chapter 4, Section 4.01). Logic propositions use the relational (comparison) operators shown in Table 12.03.

Simple condition

```
IF A < B
  THEN
    <statement(s)>
  ELSE
    <statement(s)>
ENDIF
```

Figure 12.10 Pseudocode for the selection construct

Operator	Comparison
=	Is equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to

Table 12.03 Relational operators

Conditions are either TRUE or FALSE. In pseudocode, we distinguish between the relational operator = (which tests for equality) and the assignment symbol ←.

A person is classed as a child if they are under 13 and as an adult if they are over 19. If they are between 13 and 19 inclusive they are classed as teenagers. We can write these statements as logic statements.

- If Age < 13 then person is a child.
- If Age > 19 then person is an adult.
- If Age >= 13 AND Age <= 19 then person is a teenager.

TASK 12.04

A town has a bus service where passengers under the age of 12 and over the age of 60 do not need to pay a fare. Write the logic statements for free fares.

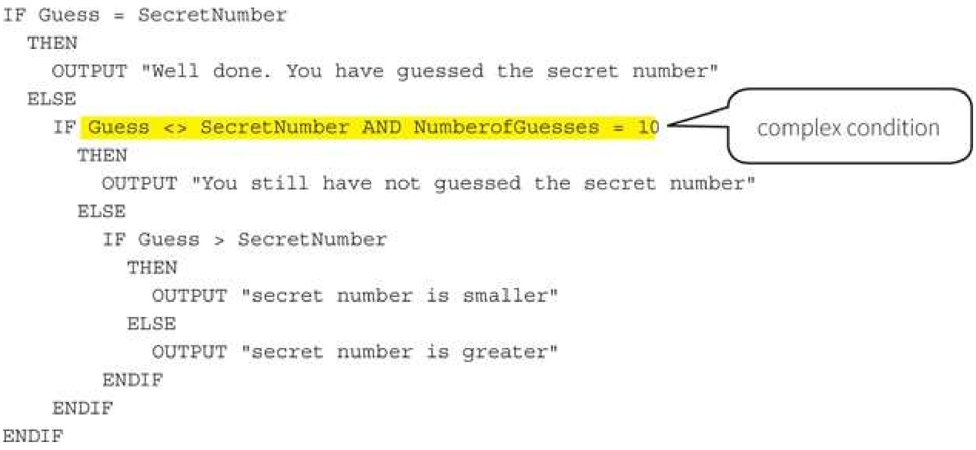
A number-guessing game follows different steps depending on certain conditions. Here is a description of the algorithm.

- The player inputs a number to guess the secret number stored.
- If the guess was correct, output a congratulations message.
- If the number input was larger than the secret number, output message “secret number is smaller”.
- If the number input was smaller than the secret number, output message “secret number is greater”.

We can re-write the number-guessing game steps as an algorithm in pseudocode:

```
SET value for secret number
INPUT Guess
IF Guess = SecretNumber
   THEN
      OUTPUT "Well done. You have guessed the secret number"
   ELSE
      IF Guess > SecretNumber
        THEN
          OUTPUT "secret number is smaller"
        ELSE
          OUTPUT "secret number is greater"
      ENDIF
ENDIF
```

More complex conditions can be formed by using the logical operators AND, OR and NOT. For example, the number-guessing game might allow the player multiple guesses; if the player has not guessed the secret number after 10 guesses, a different message is output.



```
IF Guess = SecretNumber
  THEN
    OUTPUT "Well done. You have guessed the secret number"
  ELSE
    IF Guess <> SecretNumber AND NumberofGuesses = 10
      THEN
        OUTPUT "You still have not guessed the secret number"
      ELSE
        IF Guess > SecretNumber
          THEN
            OUTPUT "secret number is smaller"
          ELSE
            OUTPUT "secret number is greater"
        ENDIF
    ENDIF
ENDIF
```

WORKED EXAMPLE 12.02

Using selection constructs

The problem to be solved: Take three numbers as input and output the largest number.

There are several different methods (algorithms) to solve this problem. Here is one method.

1. Input all three numbers at the beginning.
2. Store each of the input values in a separate variable (the identifiers are shown in Table 12.04).
3. Compare the first number with the second number and then compare the bigger one of these with the third number.
4. The bigger number of this second comparison is output.

See Worked Example 12.03 for another solution.

Identifier	Explanation

`Number1`	The first number to be input
`Number2`	The second number to be input
`Number3`	The third number to be input

Table 12.04 Identifier table for biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN
  // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE
  // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

When an `IF` statement contains another `IF` statement, we refer to these as **nested `IF` statements**.

Question 12.01

What changes do you need to make to output the smallest number?

WORKED EXAMPLE 12.03

Using selection constructs (alternative method)

The problem to be solved: Take three numbers as input and output the largest number.

This is an alternative method to Worked Example 12.02.

1. Input the first number and store it in `BiggestSoFar`
2. Input the second number and compare it with the value in `BiggestSoFar`.
3. If the second number is bigger, assign its value to `BiggestSoFar`
4. Input the third number and compare it with the value in `BiggestSoFar`
5. If the third number is bigger, assign its value to `BiggestSoFar`
6. The value stored in `BiggestSoFar` is output.

The identifiers required for this solution are shown in Table 12.05.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far

`NextNumber`	The next number to be input

Table 12.05 Identifier table for the alternative solution to the biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
OUTPUT BiggestSoFar
```

Note that when we input the third number in this method the second number gets overwritten as it is no longer needed.

There are several advantages of using the method in Worked Example 12.03 compared to the method in Worked Example 12.02.

- Only two variables are used.
- The conditional statements are not nested and do not have an ELSE part. This makes them easier to understand.
- This algorithm can be adapted more easily if further numbers are to be compared (see Worked Example 12.04).

The disadvantage of the method in Worked Example 12.03 compared to the method in Worked Example 12.02 is that there is more work involved with this algorithm. If the second number is bigger than the first number, the value of `BiggestSoFar` has to be changed. If the third number is bigger than the value in `BiggestSoFar` then the value of `BiggestSoFar` has to be changed again. Depending on the input values, this could result in two extra assignment instructions being carried out.

12.07 Loops

Look at the pseudocode algorithm in Worked Example 12.03. The two `IF` statements are identical. To compare 10 numbers, we would need to write this statement nine times. Moreover, if the problem changed to having to compare, for example, 100 numbers, our algorithm would become very tedious. If we use a repetition construct (a loop) we can avoid writing the same lines of pseudocode over and over again.

WORKED EXAMPLE 12.04

Repetition using REPEAT...UNTIL

The problem to be solved: Take 10 numbers as input and output the largest number.

We need one further variable to store a counter, so that we know when we have compared 10 numbers.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
`Counter`	Stores how many numbers have been input so far

Table 12.06 Identifier table for the biggest number problem using `REPEAT...UNTIL`

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
Counter ← 1
REPEAT
    INPUT NextNumber
    Counter ← Counter + 1
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL Counter = 10
OUTPUT BiggestSoFar
```

Note that when we input the next number in this method the previous number gets overwritten as it is no longer needed.

Question 12.02

What changes do you need to make to the algorithm in Worked Example 12.04:

a to compare 100 numbers?

b to take as a first input the number of numbers to be compared?

There is another loop construct that does the counting for us: the `FOR...NEXT` loop.

WORKED EXAMPLE 12.05

Repetition using FOR...NEXT

The problem to be solved: Take 10 numbers as input and output the largest number.

We can use the same identifiers as in Worked Example 12.04. Note that the purpose of `Counter` has changed.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far
`NextNumber`	The next number to be input
`Counter`	Counts the number of times round the loop

Table 12.07 Identifier table for biggest number problem using a `FOR` loop

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
FOR Counter ← 2 TO 10
      INPUT NextNumber
      IF NextNumber > BiggestSoFar
        THEN
          BiggestSoFar ← NextNumber
      ENDIF
NEXT Counter
OUTPUT BiggestSoFar
```

The first time round the loop, Counter is set to 2. The next time round the loop, Counter has automatically increased to 3, and so on. The last time round the loop, Counter has the value 10.

A **rogue value** is a value used to terminate a sequence of values. The rogue value is of the same data type but outside the range of normal expected values.

WORKED EXAMPLE 12.06

Repetition using a rogue value

The problem to be solved: A sequence of non-zero numbers is terminated by 0. Take this sequence as input and output the largest number.

Note: In this example the rogue value chosen is 0. It is very important to choose a rogue value that is of the same data type but outside the range of normal expected values. For example, if the input might normally include 0 then a negative value, such as −1, might be chosen.

Look at Worked Example 12.05. Instead of counting the numbers input, we need to check whether the number input is 0 to terminate the loop. The identifiers are shown in Table 12.08.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far
`NextNumber`	The next number to be input

Table 12.08 Identifier table for biggest number problem using a rogue value

A possible pseudocode algorithm is:

```
INPUT BiggestSoFar
REPEAT
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL NextNumber = 0
OUTPUT BiggestSoFar
```

This algorithm works even if the sequence consists of only one non-zero input. However, it will not work if the only input is 0. In that case, we don't want to perform the statements within the loop at all. We can use an alternative construct, the `WHILE...ENDWHILE` loop.

```
INPUT NextNumber
BiggestSoFar ← NextNumber
WHILE NextNumber <> 0 DO // sequence terminator not encountered
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
ENDWHILE
OUTPUT BiggestSoFar
```

Before we enter the loop, we check whether we have a non-zero number. To make this work for the first number, we store it in `NextNumber` and also in `BiggestSoFar`. If this first number is zero, we don't follow the instructions within the loop. For a non-zero first number this algorithm has the same effect as the algorithm using `REPEAT...UNTIL`.

WORKED EXAMPLE 12.07

Implementing the number-guessing game with a loop

Consider the number-guessing game again, this time allowing repeated guesses.

1. The player repeatedly inputs a number to guess the secret number stored.
2. If the guess is correct, the number of guesses made is output and the game stops.
3. If the number input is larger than the secret number, the player is given the message to input a smaller number.
4. If the number input is smaller than the secret number, the player is given the message to input a larger number.

The algorithm is expressed in structured English, as a flowchart and in pseudocode.

Algorithm for the number-guessing game in structured English:

```
SET value for secret number
REPEAT the following UNTIL correct guess
   INPUT guess
   count number of guesses
   COMPARE guess with secret number
   OUTPUT comment
OUTPUT number of guesses
```

We need variables to store the following values:

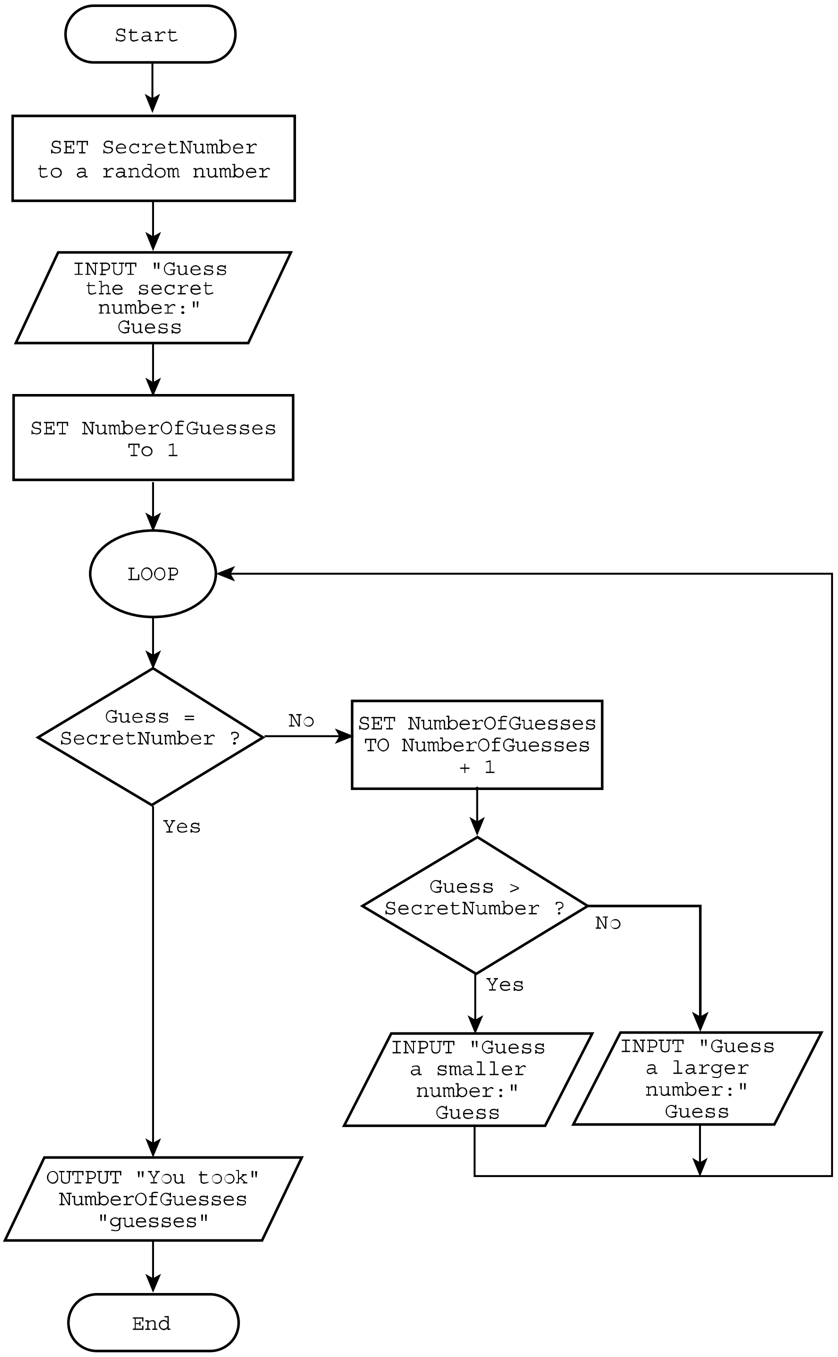
- the secret number (to be set as a random number)
- the number input by the player as a guess
- the count of how many guesses the player has made so far.

We represent this information in the identifier table shown in Table 12.09.

Identifier	Explanation
`SecretNumber`	The number to be guessed
`NumberOfGuesses`	The number of guesses the player has made
`Guess`	The number the player has input as a guess

Table 12.09 Identifier table for number-guessing game

Algorithm for the number-guessing game as a flowchart


Start
SET SecretNumber to a random number
INPUT "Guess the secret number:" Guess
SET NumberOfGuesses To 1
LOOP
Guess = SecretNumber ?
No
SET NumberOfGuesses TO NumberOfGuesses + 1
Yes
Guess > SecretNumber ?
No
Yes
INPUT "Guess a smaller number:" Guess
INPUT "Guess a larger number:" Guess
OUTPUT "You took" NumberOfGuesses "guesses"
End

Pseudocode for the number-guessing game with a post-condition loop

```
SecretNumber ← Random
NumberOfGuesses ← 0
REPEAT
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
UNTIL Guess = SecretNumber
OUTPUT NumberOfGuesses
```

Pseudocode for the number-guessing game with a pre-condition loop

```
SecretNumber ← Random
INPUT Guess
NumberOfGuesses ← 1
WHILE Guess <> SecretNumber DO
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
ENDWHILE
OUTPUT NumberOfGuesses
```

WORKED EXAMPLE 12.08

Calculating running totals and averages

The problem to be solved: Take 10 numbers as input and output the sum of these numbers and the average.

Identifier	Explanation
`RunningTotal`	Stores the sum of the numbers input so far
`Counter`	How many numbers have been input
`NextNumber`	The next number input
`Average`	The average of the numbers input

Table 12.10 Identifier table for running total and average algorithm

The following pseudocode gives a possible algorithm:

```
RunningTotal ← 0
FOR Counter ← 1 TO 10
      INPUT NextNumber
      RunningTotal ← RunningTotal + NextNumber
NEXT Counter
OUTPUT RunningTotal
Average ← RunningTotal / 10
OUTPUT Average
```

It is very important that the value stored in `RunningTotal` is initialised to zero before we start adding the numbers being input.

TIP

Which type of loop? If it is known how many repetitions are required, choose a FOR loop. If the statements inside the loop might never be executed, choose a WHILE loop. If the statements inside the loop are to be executed at least once, a REPEAT loop might be more sensible.

TASK 12.05

Change the algorithm in Worked Example 12.08 so that the sequence of numbers is terminated by a rogue value of 0.

WORKED EXAMPLE 12.09

Using nested loops

The problem to be solved: Take as input two numbers and a symbol. Output a grid made up entirely of the chosen symbol, with the number of rows matching the first number input and the number of columns matching the second number input.

For example the three input values 3, 7 and &, result in the output:

```
&&&&&&&
&&&&&&&
&&&&&&&
```

We need two variables to store the number of rows and the number of columns. We also need a variable to store the symbol. We need a counter for the rows and a counter for the columns.

Identifier	Explanation
`NumberOfRows`	Stores the number of rows of the grid
`NumberOfColumns`	Stores the number of columns of the grid
`Symbol`	Stores the chosen character symbol
`RowCounter`	Counts the number of rows
`ColumnCounter`	Counts the number of columns

Table 12.11 Identifier table for the nested loop example

```
INPUT NumberOfRows
INPUT NumberOfColumns
INPUT Symbol
FOR RowCounter ← 1 TO NumberOfRows
    FOR ColumnCounter ← 1 TO NumberOfColumns
        OUTPUT Symbol // without moving to next line
    NEXT ColumnCounter
    OUTPUT Newline   // move to the next line
NEXT RowCounter
```

Each time round the outer loop (counting the number of rows) we complete the inner loop, outputting a symbol for each count of the number of columns. This type of construct is called a **nested loop**.

12.08 Stepwise refinement

Many problems that we want to solve are bigger than the ones we met so far. To make it easier to solve a bigger problem, we break the problem down into smaller steps. These might need breaking down further until the steps are small enough to solve easily.

For a solution to a problem to be programmable, we need to break down the steps of the solution into the basic constructs of sequence, assignment, selection, repetition, input and output.

We can use a method called **stepwise refinement** to break down the steps of our outline solution into smaller steps until it is detailed enough. In Section 12.02 we looked at a recipe for a cake. The step of mixing together all the ingredients was broken down into more detailed steps.

WORKED EXAMPLE 12.10

Drawing a pyramid using stepwise refinement

The problem to be solved: Take as input a chosen symbol and an odd number. Output a pyramid shape made up entirely of the chosen symbol, with the number of symbols in the final row matching the number input.

For example the two input values A and 9 result in the following output:

```
    A
   AAA
  AAAAA
 AAAAAAA
AAAAAAAAA
```

This problem is similar to Worked Example 12.09, but the number of symbols in each row starts with one and increases by two with each row. Each row starts with a decreasing number of spaces, to create the slope effect.

Our first attempt at solving this problem using structured English is:

```
01  Set up initial values
02  REPEAT
03     Output number of spaces
04     Output number of symbols
05     Adjust number of spaces and number of symbols to be output in next row
06  UNTIL the required number of symbols have been output in one row
```

The steps are numbered to make it easier to refer to them later.

This is not enough detail to write a program in a high-level programming language. Exactly what values do we need to set?

We need as input:

- the symbol character from which the pyramid is to be formed
- the number of symbols in the final row (for the pyramid to look symmetrical, this needs to be an odd number).

We need to calculate how many spaces we need in the first row. So that the slope of the pyramid is symmetrical, this number should be half of the final row's symbols. We need to set the number of symbols to be output in the first row to 1. We therefore need the identifiers listed in Table 12.12.

Identifier	Explanation
`Symbol`	The character symbol to form the pyramid
`MaxNumberOfSymbols`	The number of symbols in the final row

`NumberOfSpaces`	The number of spaces to be output in the current row
`NumberOfSymbols`	The number of symbols to be output in the current row

Table 12.12 Identifier table for pyramid example

Using pseudocode, we now refine the steps of our first attempt. To show which step we are refining, a numbering system is used as shown.

Step 01 can be broken down as follows:

```
01   // Set up initial values expands into:
01.1 INPUT Symbol
01.2 INPUT MaxNumberOfSymbols
01.3 NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4 NumberOfSymbols ← 1
```

Remember we need an odd number for `MaxNumberOfSymbols`. We need to make sure the input is an odd number. So we further refine Step 01.2:

```
01.2   // INPUT MaxNumberOfSymbols expands into:
01.2.1 REPEAT
01.2.2    INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.2.4 // MOD 2 gives the remainder after integer division by 2
```

We can now look to refine Steps 03 and 04:

```
03 // Output number of spaces expands into:
03.1  FOR i ← 1 TO NumberOfSpaces
03.2    OUTPUT Space // without moving to next line
03.3  NEXT i
04    // Output number of symbols expands into:
04.1  FOR i ← 1 TO NumberOfSymbols
04.2    OUTPUT Symbol // without moving to next line
04.3  NEXT i
04.4  OUTPUT Newline // move to the next line
```

In Step 05 we need to decrease the number of spaces by 1 and increase the number of symbols by 2:

```
05   // Adjust values for next row expands into:
05.1 NumberOfSpaces ← NumberOfSpaces – 1
05.2 NumberOfSymbols ← NumberOfSymbols + 2
```

Step 06 essentially checks whether the number of symbols for the next row is now greater than the value input at the beginning.

```
06     UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

We can put together all the steps and end up with a solution.

```
01     // Set Values
01.1   INPUT Symbol
01.2   // Input max number of symbols (an odd number)
01.2.1 REPEAT
01.2.2     INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.3   NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4   NumberOfSymbols ← 1
02     REPEAT
03         // Output number of spaces
03.1       FOR i ← 1 TO NumberOfSpaces
```

```
03.2            OUTPUT Space // without moving to next line
03.3        NEXT i
04          // Output number of symbols
04.1        FOR i ← 1 TO NumberOfSymbols
04.2            OUTPUT Symbol // without moving to next line
04.3        NEXT i
04.4        OUTPUT Newline // move to the next line
05          // Adjust Values For Next Row
05.1        NumberOfSpaces ← NumberOfSpaces – 1
05.2        NumberOfSymbols ← NumberOfSymbols + 2
06      UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

TASK 12.06

Use stepwise refinement to output a hollow triangle. For example the two input values A and 9 result in the following output:

```
    A
   A A
  A   A
 A     A
AAAAAAAAA
```

A first attempt at solving this problem using structured English is:

```
01   Set up initial values
02   REPEAT
03       Output leading number of spaces
04       Output symbol, middle spaces, symbol
05       Adjust number of spaces and number of symbols to be output in next row
06   UNTIL the required number of symbols have been output in one row
```

12.09 Modules

Another method of developing a solution is to decompose the problem into sub-tasks. Each sub-task can be considered as a 'module' that is refined separately. Modules are procedures and functions.

A **procedure** groups together a number of steps and gives them a name (an identifier). We can use this identifier when we want to refer to this group of steps. When we want to perform the steps in a procedure we call the procedure by its name.

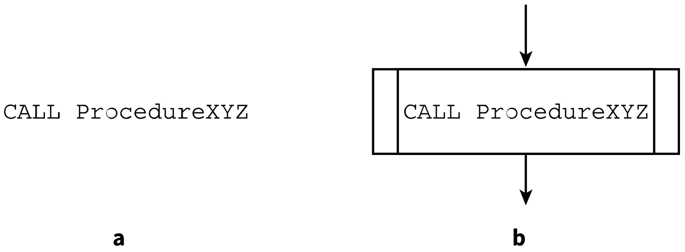


Figure 12.11 Representation of a procedure in (a) pseudocode and (b) a flowchart

A **function** groups together a number of steps and gives them a name (an identifier). These steps produce and return a value that is used in an expression. Worked Example 12.12 uses functions.

Note: Because a function returns a value, the function definition states the data type of this value. See more about data types in Chapter 13.

The rules for module identifiers are the same as for variable identifiers (see Section 12.04)

WORKED EXAMPLE 12.11

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.10.

When we want to set up the initial values, we call a procedure, using the following statement:

```
CALL SetValues
```

We can rewrite the top-level solution to our pyramid problem using a procedure for each step, as:

```
CALL SetValues
REPEAT
      CALL OutputSpaces
      CALL OutputSymbols
      CALL AdjustValuesForNextRow
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls four procedures. This means each procedure has to be defined. The procedure definitions are:

```
PROCEDURE SetValues
      INPUT Symbol
      CALL InputMaxNumberOfSymbols // need to ensure it is an odd number
      NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
      NumberOfSymbols ← 1
ENDPROCEDURE
PROCEDURE InputMaxNumberOfSymbols
      REPEAT
          INPUT MaxNumberOfSymbols
      UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE
PROCEDURE OutputSpaces
      FOR Count1 ← 1 TO NumberOfSpaces
```

```
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
PROCEDURE AdjustValuesForNextRow
    NumberOfSpaces ← NumberOfSpaces – 1
    NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

TASK 12.07

Amend your algorithm for Task 12.05 to use modules.

WORKED EXAMPLE 12.12

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.11.

We can rewrite the top-level solution to our pyramid problem using procedures and functions.

```
01    CALL SetValues
02    REPEAT
03         CALL OutputSpaces
04         CALL OutputSymbols
05.1       NumberOfSpaces ← AdjustedNumberOfSpaces
05.2       NumberOfSymbols ← AdjustedNumbeOfSymbols
06    UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls three procedures. It also makes use of two functions in lines 05.1 and 05.2.

The procedures and functions have to be defined.

```
PROCEDURE SetValues
    INPUT Symbol
    MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
    RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
```

```
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
       OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumberOfSymbols RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that procedure `SetValues` uses a function `ValidatedMaxNumberOfSymbols`.

One benefit of using modules is that individual modules can be reused in other solutions. Therefore, modules should be designed to be self-contained. That means they should not rely on external variables. All variables that are required by a module should be passed to it using parameters. To illustrate this, look at Worked Example 12.13

WORKED EXAMPLE 12.13

Drawing a pyramid using modules and parameters

The problem is the same as in Worked Example 12.12.

```
CALL SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
REPEAT
     CALL OutputSpaces(NumberOfSpaces)
     CALL OutputSymbols(NumberOfSymbols, Symbol)
     NumberOfSpaces ← AdjustedNumberOfSpaces(NumberOfSpaces)
     NumberOfSymbols ← AdjustedNumbeOfSymbols(NumberOfSymbols)
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

Module definitions:

```
PROCEDURE SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
     INPUT Symbol
     MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
     NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
     NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
     REPEAT
         INPUT MaxNumberOfSymbols
     UNTIL MaxNumberOfSymbols MOD 2 = 1
     RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces(NumberOfSpaces)
     FOR Count1 ← 1 TO NumberOfSpaces
         OUTPUT Space // without moving to next line
     NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
```

```
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces(NumberOfSpaces) RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumbeOfSymbols(NumberOfSymbols) RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that the procedure `OutputSpaces` uses a variable, `Count1`, which is used only within the module. Similarly, `OutputSymbols` uses variable `Count2` only within the module. We call such a variable a **local variable** (see Chapter 14, Section 14.09). A variable available to all modules is known as a **global variable** (see Chapter 14, Section 14.09).

TIP

Good design uses local variables as it makes modules independent and re-usable.

Reflection Point:

Can you think of other problems and use decomposition to break them down into basic constructs, input and output statements?

Summary

- Abstraction involves filtering out information that is not needed to solve the problem.
- Decomposition is breaking down problems into sub-problems, leading to the concept of a program module.
- An algorithm is a sequence of steps that can be carried out to solve a problem.
- Algorithms are expressed using the four basic constructs of assignment, sequence, selection and repetition.
- Algorithms can be documented using pseudocode.
- Stepwise refinement: breaking down the steps of an outline solution into smaller and smaller steps.
- Logic statements use the relational operators =, <, >, <>, <= and >= and the logic operators AND, OR and NOT.
- Selection constructs and conditional loops use conditions to determine the steps to be followed.

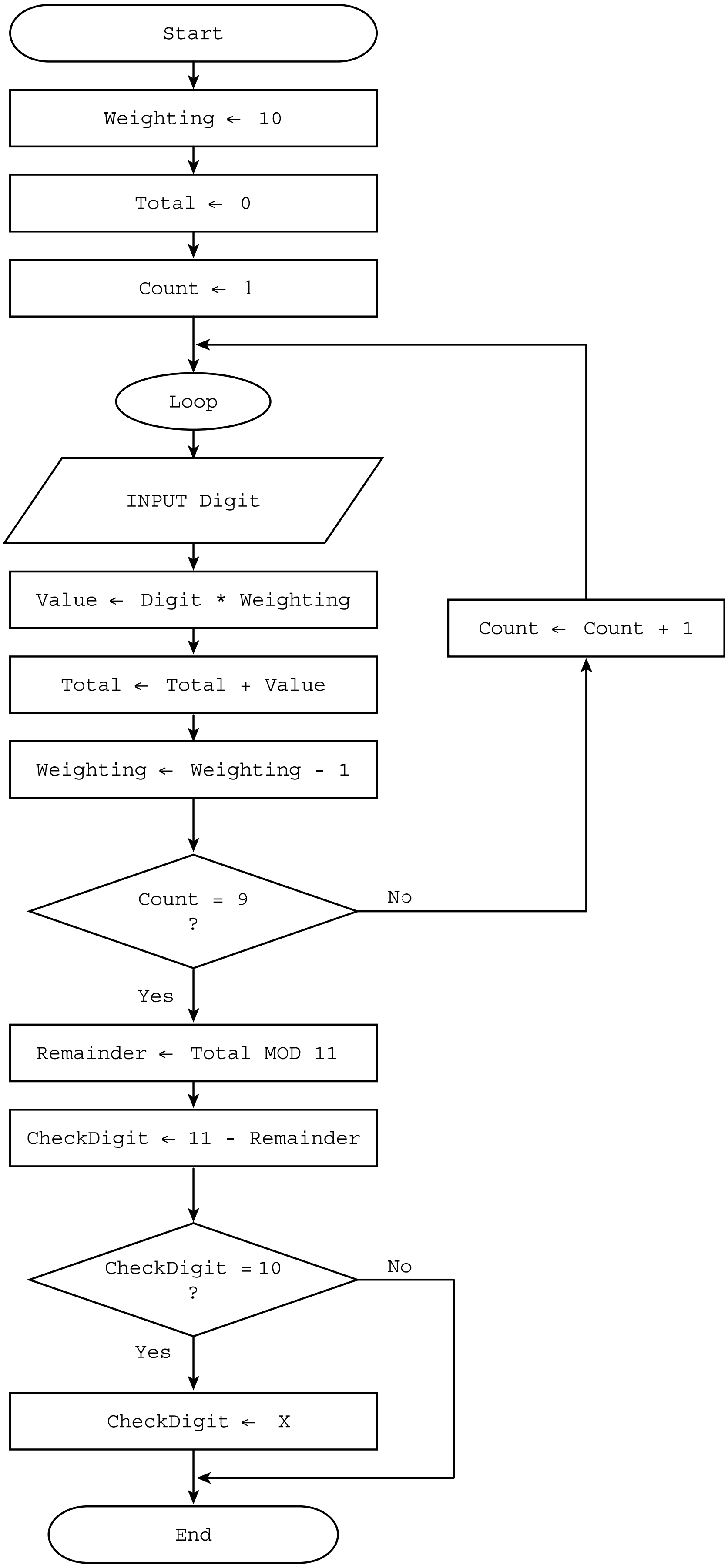
Exam-style Questions

1 The Modulo-11 method of calculating a check digit for a sequence of nine digits is as follows:

Each digit in the sequence is given a weight depending on its position in the sequence. The leftmost digit has a weight of 10. The next digit to the right has a weight of 9, the next one 8 and so on. Values are calculated by multiplying each digit by its weight. These values are added together and the sum is divided by 11. The remainder from this division is subtracted from 11 and this value is the check digit. If this value is 10, then the check digit is X. Note that `x MOD y` gives the remainder from the division of `x` by `y`.

The flowchart shows the algorithm for calculating the Modulo-11 check digit.

Write pseudocode from the flowchart.

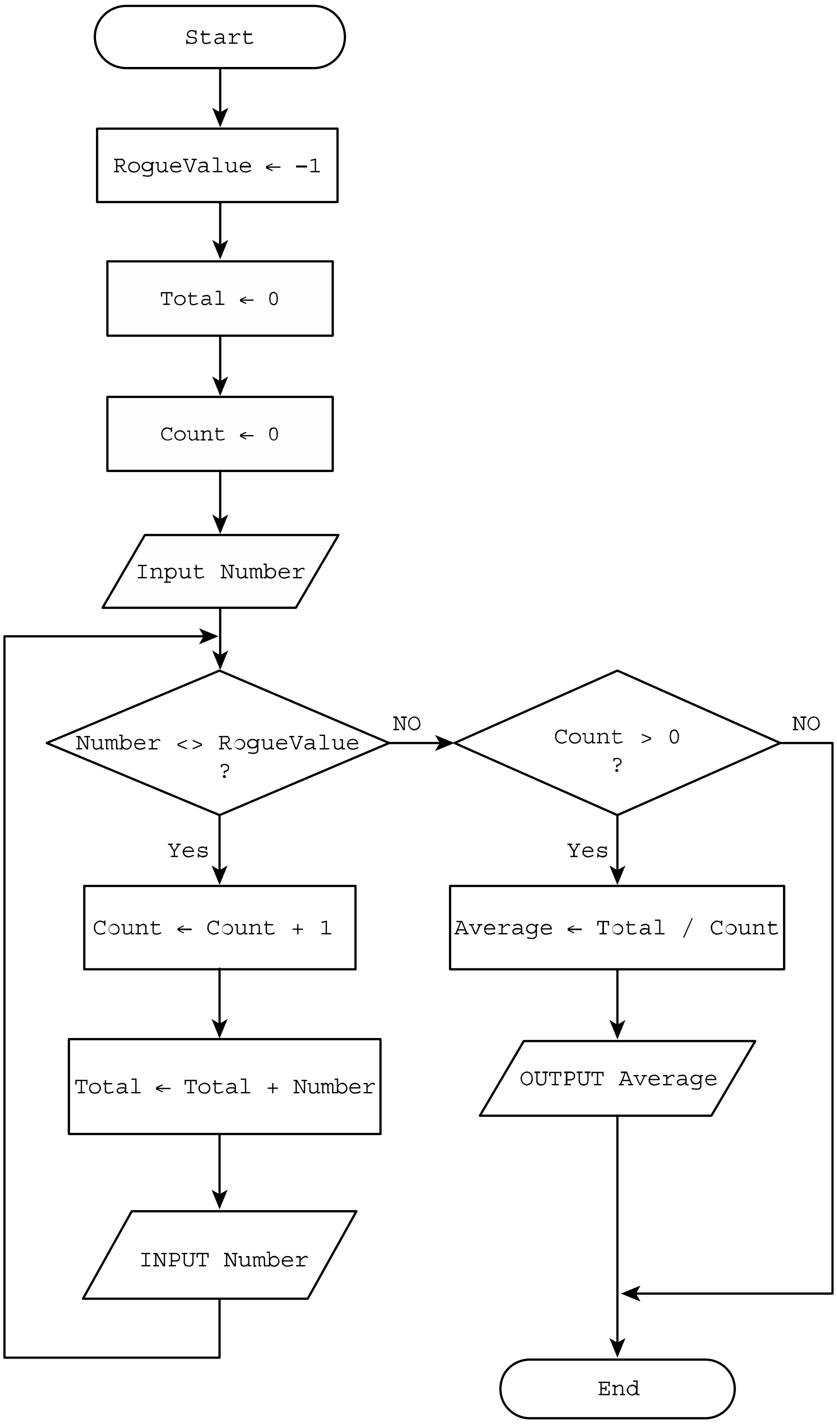

Start
Weighting ← 10
Total ← 0
Count ← 1
Loop
INPUT Digit
Value ← Digit * Weighting
Total ← Total + Value
Weighting ← Weighting - 1
Count = 9
?
No
Count ← Count + 1
Yes
Remainder ← Total MOD 11
CheckDigit ← 11 - Remainder
CheckDigit = 10
?
No
Yes
CheckDigit ← X
End

[9]

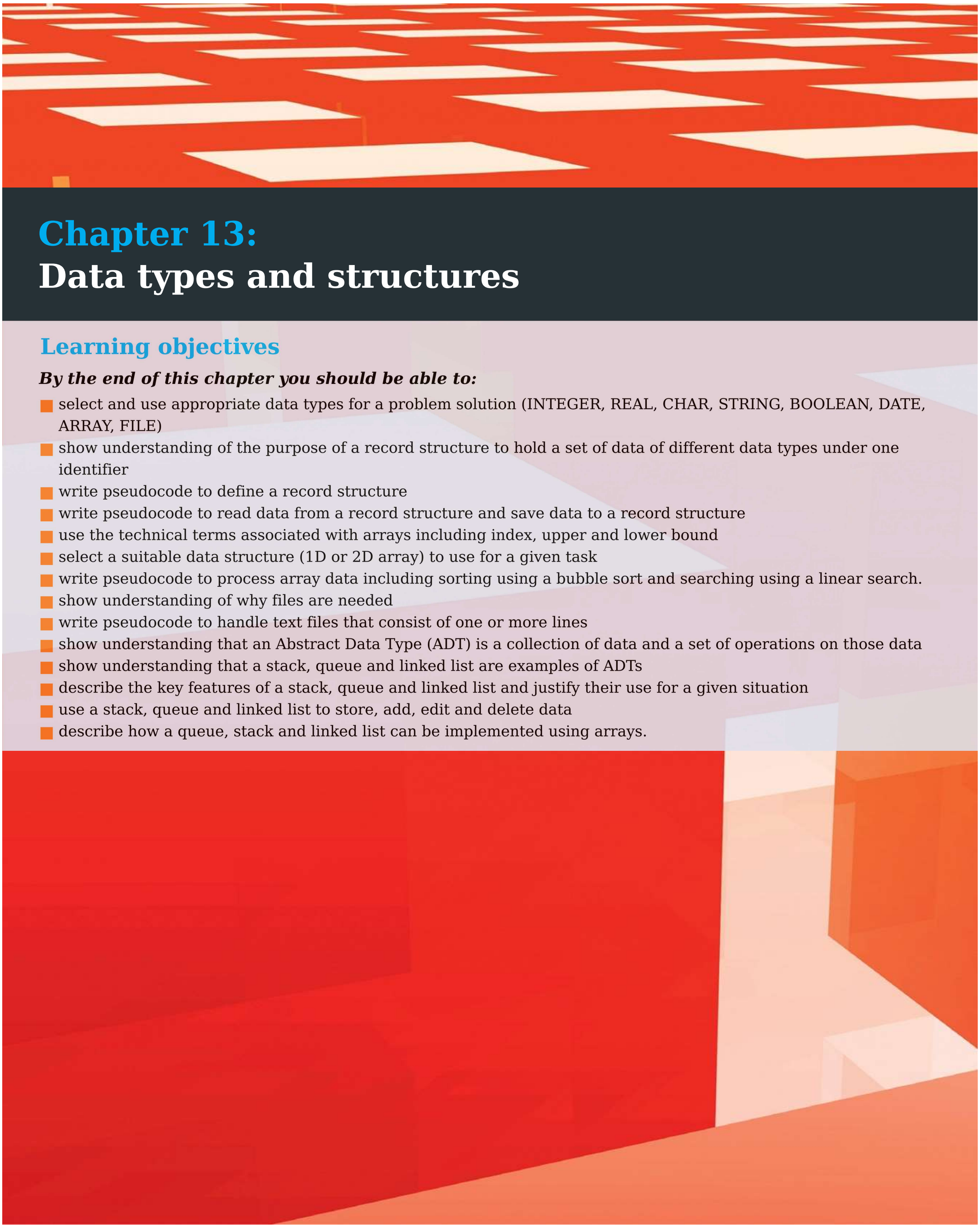
2 Write pseudocode for the following problem given in structured English.

```
REPEAT the following UNTIL the number input is zero
INPUT a number
Check whether number is positive or negative
Increment positive number count if the number is positive
Increment negative number count if the number is negative
```
[7]

3 Write pseudocode from the given flowchart. Use a `WHILE` loop.


Start
RogueValue ← −1
Total ← 0
Count ← 0
Input Number
Number <> RogueValue ?
NO
Yes
Count > 0 ?
NO
Yes
Count ← Count + 1
Total ← Total + Number
INPUT Number
Average ← Total / Count
OUTPUT Average
End

[8]



Chapter 13:

Data types and structures

Learning objectives

By the end of this chapter you should be able to:

- select and use appropriate data types for a problem solution (INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE, ARRAY, FILE)
- show understanding of the purpose of a record structure to hold a set of data of different data types under one identifier
- write pseudocode to define a record structure
- write pseudocode to read data from a record structure and save data to a record structure
- use the technical terms associated with arrays including index, upper and lower bound
- select a suitable data structure (1D or 2D array) to use for a given task
- write pseudocode to process array data including sorting using a bubble sort and searching using a linear search.
- show understanding of why files are needed
- write pseudocode to handle text files that consist of one or more lines
- show understanding that an Abstract Data Type (ADT) is a collection of data and a set of operations on those data
- show understanding that a stack, queue and linked list are examples of ADTs
- describe the key features of a stack, queue and linked list and justify their use for a given situation
- use a stack, queue and linked list to store, add, edit and delete data
- describe how a queue, stack and linked list can be implemented using arrays.

13.01 Data types

Primitive data types

In Chapter 12 we used variables to store values required by our algorithm. Look at Worked Example 12.01. The Identifier Table 12.02 lists two variable identifiers: `Miles` and `Km`. An identifier table should also show what sort of data (or data type) is going to be stored in each variable. The explanation shows that Miles will be a whole number, but that Km will be calculated using the formula Miles * 1.61. This will result in a number that may not be a whole number.

Primitive data types are those variables that can be defined simply by commands built into the programming language. Primitive data types are also known as atomic data types. In Computer Science a whole number is referred to as an INTEGER and a number with a decimal point is referred to as a REAL. Conditions are either TRUE or FALSE. These are logical values known as BOOLEAN. Sometimes we may want to store a single character; this is referred to as a CHAR.

A value that will always be a whole number should be defined to be of type INTEGER, such as when counting the iterations of a loop.

TIP

See Table 13.01 for a list of data types you should be familiar with.

See Chapter 1 (Sections 1.02 and 1.03) on how integers and characters are represented inside the computer. Chapter 16 (Section 16.03) covers the internal representation of real (single, double, float) numbers.

Further data types

If we want to store several characters; this is known as a string.

Note that there is a difference between the number 12 and the string “12”.

The string data type is known as a structured type because it is essentially a sequence of characters. A special case is the empty string: a value of data type string, but with no characters stored in it.

When we write a date, such as 3 February 2018, we can also write this as a set of three numbers: 3/2/2018. Sometimes we might wish to calculate with dates, such as taking one date away from another to find out how many days, months and years are between these dates. To make it easier to do this, DATE has been designed as a data type. To see how different programming languages implement this data type, see Chapter 14 Section 14.03.

INTEGER	A signed whole number
REAL	A signed number with a decimal point
CHAR	A single character
STRING	A sequence of zero or more characters
BOOLEAN	The logical values TRUE and FALSE
DATE	A date consisting of day, month and year, sometimes including a time in hours, minutes and seconds

Table 13.01 Summary of pseudocode data types

TASK 13.01

Look at the identifier tables in Chapter 12 (Tables 12.06 and 12.09 to 12.12). Give the data type that is appropriate for each variable listed.

13.02 The record type

Sometimes variables of different data types are a logical group, such as data about a person (name, date of birth, height, number of siblings, whether they are a full-time student).

Name is a STRING; date of birth is a DATE; height is a REAL; number of siblings is an INTEGER; whether they are a full-time student is a BOOLEAN.

We can declare a record type to suit our purposes. The record type is known as a user-defined type, because the programmer can decide which variables (fields) to include as a record.

TIP

A record type is also known as a composite type.

In pseudocode a record type is declared as:

```
TYPE <TypeIdentifier>
    DECLARE <field identifier> : <data type>
    .
    .
ENDTYPE
```

We can now declare a variable of this record type:

```
DECLARE <variable identifier> : <record type>
```

And then access an individual field using the dot notation:

```
<variable identifier>.<field identifier>
```

Using the example above we can declare a Person record type:

```
TYPE PersonType
    Name : STRING
    DateOfBirth : DATE
    Height : REAL
    NumberOfSiblings : INTEGER
    IsFullTimeStudent : BOOLEAN
ENDTYPE
```

To declare a variable of this type we write:

```
DECLARE Person : PersonType
```

And now we can assign a value to a field of this Person record:

```
Person.Name ← "Fred"
Person.NumberOfSiblings ← 3
Person.IsFullTimeStudent ← TRUE
```

To output a field of a record:

```
OUTPUT Person.Name
```

TASK 13.02

Write the declaration of a record type to store the details of a book: Title, Year of publication, Price, ISBN.

Write the statements required to assign the values “Computer Science”, 2019, £44.95, “9781108733755” to the fields respectively.

13.03 Arrays

Sometimes we want to organise data values into a list or a table / matrix. In most programming languages these structures are known as arrays. An array is an ordered set of data items, usually of the same type, grouped together using a single identifier. Individual array elements are addressed using an **array index** for each array dimension.

A list is a one-dimensional (1D) array and a table or matrix is a two-dimensional (2D) array.

TIP

When writing pseudocode, arrays need to be declared before they are used. This means choosing an identifier, the data type of the values to be stored in the array and **upper bound** and **lower bound** for each dimension.

13.04 One-dimensional arrays

When we write a list on a piece of paper and number the individual items, we would normally start the numbering with 1. You can view a 1D array like a numbered list of items. Many programming languages number array elements from 0 (the lower bound), including VB.NET, Python and Java. Depending on the problem to be solved, it might make sense to ignore element 0. The upper bound is the largest number used for numbering the elements of an array.

In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Here is a pseudocode example:

```
DECLARE List1 : ARRAY[1:3] OF STRING  // 3 elements in this list
DECLARE List2 : ARRAY[0:5] OF INTEGER // 6 elements in this list
DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
DECLARE List4 : ARRAY[0:25] OF CHAR  // 26 elements in this list
```

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
<arrayIdentifier>[x]
```

The *n*th element within the array `MyList` is referred to as `MyList[n]`.

Here is a pseudocode example:

```
NList[25] ← 0  // set 25th element to zero
AList[3] ← 'D' // set 3rd element to letter D
```

WORKED EXAMPLE 13.01

Working with a one-dimensional array

The problem to be solved: Take seven numbers as input and store them for later use.

We could use seven separate variables. However, if we wanted our algorithm to work with 70 numbers, for example, then setting up 70 variables would be complicated and waste time. Instead, we can make use of a data structure known as a ‘linear list’ or a 1D array.

This array is given an identifier, for example `MyList`, and each element within the array is referred to using this identifier and its position (index) within the array. For example, `MyList[4]` refers to the element at position 4 in the `MyList` array. If we are counting the element at position 0 as the first element, `MyList[4]` refers to the fifth element.

We can use a loop to access each array element in turn. If the numbers input to the pseudocode algorithm below are 25, 34, 98, 7, 41, 19 and 5 then the algorithm will produce the result in Figure 13.01.

```
FOR Index ← 0 TO 6
    INPUT MyList[Index]
NEXT Index
```

Index	**MYList**
[0]	25
[1]	34
[2]	98
[3]	7
[4]	41
[5]	19
[6]	5

Figure 13.01 `Mylist` array populated by a loop

TASK 13.03

Define two arrays, one for your friends’ names and one for their ages as shown in Figure 13.02.

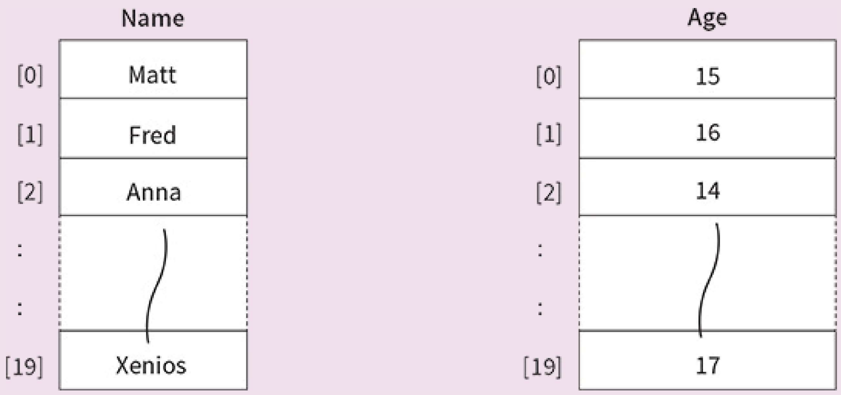


Figure 13.02 Arrays for names and ages

WORKED EXAMPLE 13.02

Searching a 1D array

The problem to be solved: Take a number as input. Search for this number in an existing 1D array of seven numbers (see Worked Example 13.01).

Start at the first element of the array and check each element in turn until the search value is found or the end of the array is reached. This method is called a **linear search**.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The number of elements in the array
`SearchValue`	`INTEGER`	The value to be searched for
`Found`	`BOOLEAN`	`TRUE` if the value has been found `FALSE` if the value has not been found
`Index`	`INTEGER`	Index of the array element currently being processed

Table 13.02 Identifier table for linear search algorithm

```
MaxIndex ← 6
INPUT SearchValue
Found ← FALSE
Index ← –1
REPEAT
    Index ← Index + 1
    IF MyList[Index] = SearchValue
      THEN
        Found ← TRUE
    ENDIF
UNTIL FOUND = TRUE OR Index >= MaxIndex
IF Found = TRUE
```

```
  THEN
    OUTPUT "Value found at location: " Index
  ELSE
    OUTPUT "Value not found"
ENDIF
```

The complex condition to the `REPEAT...UNTIL` loop allows us to exit the loop when the search value is found. Using the variable `Found` makes the algorithm easier to understand. `Found` is initialised (first set) to `FALSE` before entering the loop and set to `TRUE` if the value is found.

If the value is not in the array, the loop terminates when `Index` is greater than or equal to `MaxIndex`. That means we have come to the end of the array. Note that using `MaxIndex` in the logic statement to terminate the loop makes it much easier to adapt the algorithm when the array consists of a different number of elements. The algorithm only needs to be changed in the first line, where `MaxIndex` is given a value.

TASK 13.04

Use the algorithm in Worked Example 13.02 as a design pattern. Write an algorithm using the arrays from Task 13.03 to search for a friend's name and output their age.

WORKED EXAMPLE 13.03

Sorting elements in a 1D array

The simplest way to sort an unordered list of values is the following method.

1. Compare the first and second values. If the first value is larger than the second value, swap them.
2. Compare the second and third values. If the second value is larger than the third value, swap them.
3. Compare the third and fourth values. If the third value is larger than the fourth value, swap them.
4. Keep on comparing adjacent values, swapping them if necessary, until the last two values in the list have been processed.

Figure 13.03 shows what happens to the values as we work down the array, following this algorithm.

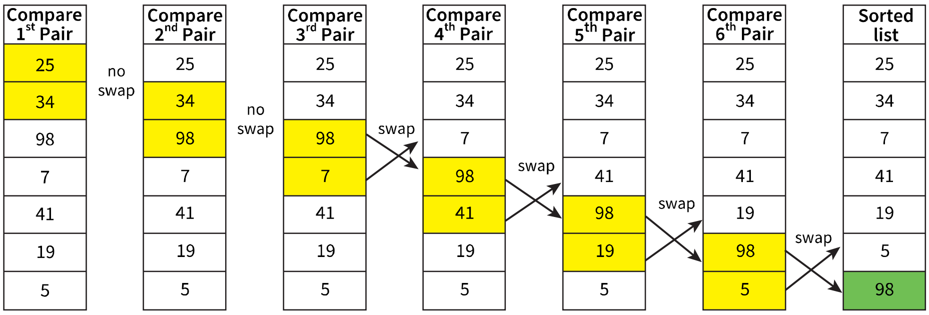
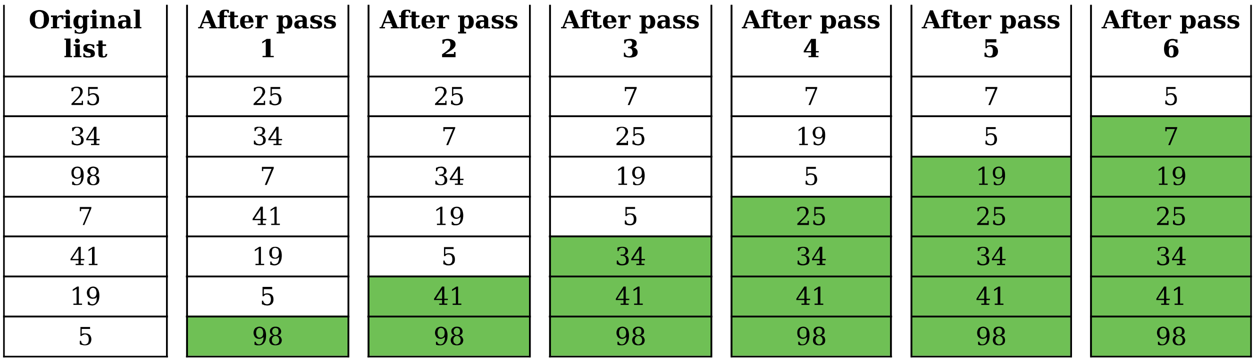

Compare 1st Pair
25
34
98
7
41
19
5
no swap
Compare 2nd Pair
25
34
98
7
41
19
5
no swap
Compare 3rd Pair
25
34
98
7
41
19
5
swap
Compare 4th Pair
25
34
7
98
41
19
5
swap
Compare 5th Pair
25
34
7
41
98
19
5
swap
Compare 6th Pair
25
34
7
41
19
98
5
swap
Sorted list
25
34
7
41
19
5
98

Figure 13.03 Swapping values working down the array

When we have completed the first pass through the entire array, the largest value is in the correct position at the end of the array. The other values may or may not be in the correct order.

We need to work through the array again and again. After each pass through the array the next largest value will be in its correct position, as shown in Figure 13.04.



Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
25	25	25	7	7	7	5
34	34	7	25	19	5	7
98	7	34	19	5	19	19
7	41	19	5	25	25	25
41	19	5	34	34	34	34
19	5	41	41	41	41	41
5	98	98	98	98	98	98

Figure 13.04 States of the array after each pass

In effect we perform a loop within a loop, a nested loop. This method is known as a **bubble sort**. The name comes from the fact that smaller values slowly rise to the top, like bubbles in a liquid.

The identifiers needed for the algorithm are listed in Table 13.03.

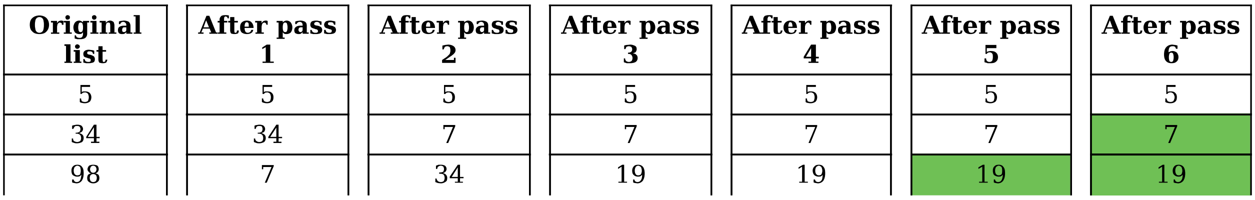
Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`i`	`INTEGER`	Counter for outer loop
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

Table 13.03 Identifier table for bubble sort algorithm

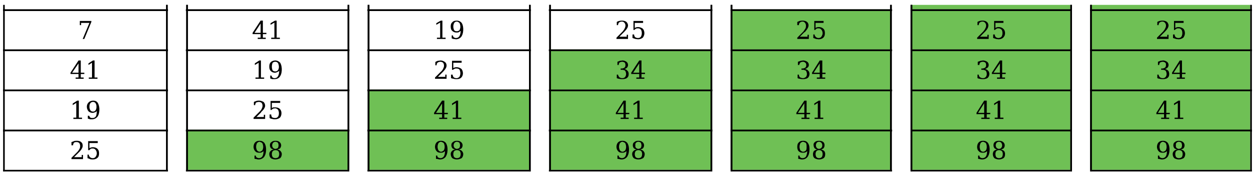
The algorithm in pseudocode is:

```
n ← MaxIndex – 1
FOR i ← 0 TO MaxIndex – 1
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
        ENDIF
    NEXT j
    n ← n – 1 // this means the next time round the inner loop, we don't
              // look at the values already in the correct positions.
NEXT i
```

The values to be sorted may already be in the correct order before the outer loop has been through all its iterations. Look at the list of values in Figure 13.05. It is only slightly different from the first list we sorted.



Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
5	5	5	5	5	5	5
34	34	7	7	7	7	7
98	7	34	19	19	19	19



7	41	19	25	25	25	25
41	19	25	34	34	34	34
19	25	41	41	41	41	41
25	98	98	98	98	98	98

Figure 13.05 States of the list after each pass

After the third pass the values are all in the correct order but our algorithm will carry on with three further passes through the array. This means we are making comparisons when no further comparisons need to be made.

If we have gone through the whole of the inner loop (one pass) without swapping any values, we know that the array elements must be in the correct order. We can therefore replace the outer loop with a conditional loop.

We can use a variable `NoMoreSwaps` to store whether or not a swap has taken place during the current pass. We initialise the variable `NoMoreSwaps` to `TRUE`. When we swap a pair of values we set `NoMoreSwaps` to `FALSE`. At the end of the pass through the array we can check whether a swap has taken place.

The identifier table for this improved algorithm is shown in Table 13.04.

Identifier	**Data type**	**Explanation**
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`NoMoreSwaps`	`BOOLEAN`	`TRUE` when no swaps have occurred in current pass `FALSE` when a swap has occurred
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

Table 13.04 Identifier table for improved bubble sort algorithm

This improved algorithm in pseudocode is:

```
n ← MaxIndex – 1
REPEAT
    NoMoreSwaps ← TRUE
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
            NoMoreSwaps ← FALSE
        ENDIF
    NEXT j
    n ← n – 1
UNTIL NoMoreSwaps = TRUE
```

Discussion Point:

What happens if the array elements are already in the correct order?

TASK 13.05

Rewrite the algorithm in Worked Example 13.03 to sort the array elements into descending order.

13.05 Two-dimensional arrays

When we write a table of data (a matrix) on a piece of paper and want to refer to individual elements of the table, the convention is to give the row number first and then the column number. When declaring a 2D array, the number of rows is given first, then the number of columns. Again we have lower and upper bounds for each dimension.

In pseudocode, a 2D array declaration is written as:

```
DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>,
<lBound2>:<uBound2>] OF <dataType>
```

The array elements in a 2D array can be numbered from 0. Sometimes it is more intuitive to use rows from row 1 and columns from column 1, as shown with the board game in Worked Example 13.05.

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

```
Board : ARRAY[1:6,1:7] OF INTEGER
```

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

```
<arrayIdentifier>[x,y]
```

Pseudocode example:

```
Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

When we want to access each element of a 1D array, we use a loop to access each element in turn. When working with a 2D array, we need a loop to access each row. Within each row we need to access each column. This means we use a loop within a loop (nested loops).

In structured English our algorithm is:

```
For each row
   For each column
      Assign the initial value to the element at the current position
```

WORKED EXAMPLE 13.04

Working with two-dimensional arrays and nested loops

Using pseudocode, the algorithm to set each element of array `ThisTable` to zero is:

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        ThisTable[Row, Column] ← 0
    NEXT Column
NEXT Row
```

We need the identifiers shown in Table 13.05.

Identifier	Data type	Explanation
`ThisTable`	`ARRAY[0:3, 0:5] OF INTEGER`	Table data structure (2D array) to store values
`MaxRowIndex`	`INTEGER`	The upper bound of the row index
`MaxColumnIndex`	`INTEGER`	The upper bound of the column index
`Row`	`INTEGER`	Counter for the row index
`Column`	`INTEGER`	Counter for the column index

Table 13.05 Identifier table for working with a table

When we want to output the contents of a 2D array, we again need nested loops. We want to output all the values in one row of the array on the same line. At the end of the row, we want to output a new line.

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        OUTPUT ThisTable[Row, Column] // stay on same line
    NEXT Column
    OUTPUT Newline      // move to next line for next row
NEXT Row
```

TASK 13.06

1. Declare a 2D array to store the board data for the game Noughts and Crosses. The empty squares of the board are to be represented by a space. Player A's counters are to be represented by "O". Player B's counters are to be represented by "X".
2. Initialise the array to start with each square being empty.
3. Write a statement to represent player A placing their counter in the top left square.
4. Write a statement to represent player B placing their counter in the middle square.

WORKED EXAMPLE 13.05

Creating a program to play Connect 4

Connect 4 is a game played by two players. In the commercial version shown in Figure 13.06, one player uses red tokens and the other uses black. Each player has 21 tokens. The game board is a vertical grid of six rows and seven columns.

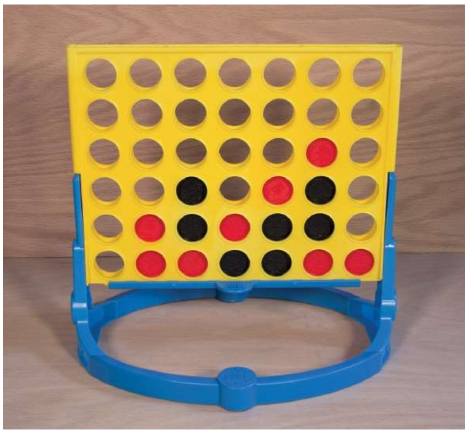


Figure 13.06 A Connect 4 board

Columns get filled with tokens from the bottom. The players take it in turns to choose a column that is not full and drop a token into this column. The token will occupy the lowest empty position in the chosen column. The winner is the player who is the first to connect four of their own tokens in a horizontal, vertical or diagonal line. If all tokens have been used and neither player has connected four tokens, the game ends in a draw.

If we want to write a program to play this game on a computer, we need to work out the steps required to 'solve the problem', that means to let players take their turn in placing tokens and checking for a winner. We will designate our players (and their tokens) by 'O' and 'X'. The game board will be represented by a 2D array. To simplify the problem, the winner is the player who is the first to connect four of their tokens horizontally or vertically.

Our first attempt in structured English is:

```
Initialise board
Set up game
Display board
While game not finished
   Player makes a move
   Display board
   Check if game finished
   If game not finished, swap player
```

The top-level pseudocode version using modules is:

```
01 CALL InitialiseBoard
02 CALL SetUpGame
03 CALL OutputBoard
04 WHILE GameFinished = FALSE DO
05     CALL PlayerMakesMove
06     CALL OutputBoard
07     CALL CheckGameFinished
08     IF GameFinished = FALSE
09       THEN
10         CALL SwapThisPlayer
11     ENDIF
12 ENDWHILE
```

Note that Steps 03 and 06 are the same. This means that we can save ourselves some effort. We only need to define this module once, but can call it from more than one place. This is one of the advantages of using modules.

The identifier table for the program is shown in Table 13.06.

Identifier	**Data type**	**Explanation**
`Board`	`ARRAY[1:6,1:7] OF CHAR`	2D array to represent the board
`InitialiseBoard`		Procedure to initialise the board to all blanks
`SetUpGame`		Procedure to set initial values for `GameFinished` and `ThisPlayer`
`GameFinished`	`BOOLEAN`	FALSE if the game is not finished TRUE if the board is full or a player has won
`ThisPlayer`	`CHAR`	`'O'` when it is Player O's turn `'X'` when it is Player X's turn
`OutputBoard`		Procedure to output the current contents of the board
`PlayerMakesMove`		Procedure to place the current player's token into the chosen board location
`CheckGameFinished`		Procedure to check if the token just placed makes the current player a winner or board is full
`SwapThisPlayer`		Procedure to change player's turn

Table 13.06 Initial identifier table for Connect 4 game

Now we can refine each procedure (module). This is likely to add some more identifiers to our identifier table. The additional entries required are shown after each procedure.

```
PROCEDURE InitialiseBoard
```

```
    FOR Row ← 1 TO 6
        FOR Column ← 1 TO 7
            Board[Row, Column] ← BLANK // use a suitable value for blank
        NEXT Column
    NEXT Row
ENDPROCEDURE
```

Identifier	Data type	Explanation
Row	INTEGER	Loop counter for the rows
Column	INTEGER	Loop counter for the columns
BLANK	CHAR	A value that represents an empty board location

Table 13.07 Additional identifiers for the `InitialiseBoard` procedure

```
PROCEDURE SetUpGame
    ThisPlayer ← 'O' // Player O always starts
    GameFinished ← FALSE
ENDPROCEDURE
PROCEDURE OutputBoard
    FOR Row ← 6 DOWNTO 1
        FOR Column ← 1 TO 7
            OUTPUT Board[Row, Column] // don't move to next line
        NEXT Column
        OUTPUT Newline // move to next line
    NEXT Row
ENDPROCEDURE
PROCEDURE PlayerMakesMove
    ValidColumn ← PlayerChoosesColumn // a module returns column number
    ValidRow ← FindFreeRow // a module returns row number
    Board[ValidRow, ValidColumn] ← ThisPlayer
ENDPROCEDURE
```

Identifier	Data type	Explanation
ValidColumn	INTEGER	The column number the player has chosen
PlayerChoosesColumn	INTEGER	Function to get the current player’s valid choice of column
ValidRow	INTEGER	The row number that represents the first free location in the chosen column
FindFreeRow	INTEGER	Function to find the next free location in the chosen column

Table 13.08 Additional identifiers for the `PlayerMakesMove` procedure

```
FUNCTION PlayerChoosesColumn RETURNS INTEGER// returns a valid column number
    OUTPUT "Player ", ThisPlayer, "'s turn."
    REPEAT
        OUTPUT "Enter a valid column number: "
        INPUT ColumnNumber
    UNTIL ColumnNumberValid = TRUE // check whether the column number is valid
    RETURN ColumnNumber
ENDFUNCTION
```

Identifier	Data type	Explanation
`ColumnNumber`	`INTEGER`	The column number chosen by the current player
`ColumnNumberValid`	`BOOLEAN`	Function to check whether the chosen column is valid

Table 13.09 Additional identifiers for `PlayerChoosesColumn` function

Note that we need to define the function `ColumnNumberValid`. A column is valid if it is within the range 1 to 7 inclusive and there is still at least one empty location in that column.

```
FUNCTION ColumnNumberValid RETURNS BOOLEAN
  // returns whether or not the column number is valid
    Valid ← FALSE
    IF ColumnNumber >= 1 AND ColumnNumber <= 7
      THEN
        IF Board[6, ColumnNumber] = BLANK // at least 1 empty space in column
          THEN
            Valid ← TRUE
        ENDIF
    ENDIF
    RETURN Valid
ENDFUNCTION
```

Identifier	Data type	Explanation
`Valid`	`BOOLEAN`	`FALSE` if column number is not valid `TRUE` if column number is valid

Table 13.10 Additional identifier for the `ColumnNumberValid` function

```
FUNCTION FindFreeRow RETURNS INTEGER
  // returns the next free position
    ThisRow ← 1
    WHILE Board[ThisRow, ValidColumn] <> BLANK DO // find first empty cell
        ThisRow ← ThisRow + 1
    ENDWHILE
    RETURN ThisRow
ENDFUNCTION
```

Identifier	Data type	Explanation
`ThisRow`	`INTEGER`	Points to the next row to be checked

Table 13.11 Additional identifier for the `FindFreeRow` function

```
PROCEDURE CheckGameFinished
    WinnerFound ← FALSE
    CALL CheckIfPlayerHasWon
    IF WinnerFound = TRUE
      THEN
        GameFinished ← TRUE
        OUTPUT ThisPlayer " is the winner"
      ELSE
        CALL CheckForFullBoard
    ENDIF
ENDPROCEDURE
```

Note that the `CheckGameFinished` procedure uses two further procedures that we need to define.

Identifier	Data type	Explanation
`WinnerFound`	`BOOLEAN`	`FALSE` if no winning line `TRUE` if a winning line is found
`CheckIfPlayerHasWon`		Procedure to check if there is a winning line
`CheckVerticalLineInValidColumn`		Procedure to check if there is a winning vertical line in the column the last token was placed in
`CheckForFullBoard`		Procedure to check whether the board is full

Table 13.12 Additional identifiers for the `CheckGameFinished` procedure

```
PROCEDURE CheckIfPlayerHasWon
    WinnerFound ← False
    CALL CheckHorizontalLine
    IF WinnerFound = FALSE
      THEN
        CALL CheckVerticalLine
ENDPROCEDURE
PROCEDURE CheckHorizontalLine
    FOR i ← 1 TO 4
       IF Board[ValidRow, i] = ThisPlayer AND
         Board[ValidRow, i + 1] = ThisPlayer AND
         Board[ValidRow, i + 2] = ThisPlayer AND
         Board[ValidRow, i + 3] = ThisPlayer
         THEN
           WinnerFound ← TRUE
       ENDIF
    NEXT i
ENDPROCEDURE
PROCEDURE CheckVerticalLine
    IF ValidRow = 4 OR ValidRow = 5 OR ValidRow = 6
      THEN
        IF Board[ValidRow, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 1, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 2, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 3, ValidColumn] = ThisPlayer
          THEN
            WinnerFound ← TRUE
        ENDIF
    ENDIF
ENDPROCEDURE
PROCEDURE CheckForFullBoard
    BlankFound ← FALSE
    ThisRow ← 0
    REPEAT
        ThisColumn ← 0
        ThisRow ← ThisRow + 1
```

```
            REPEAT
                ThisColumn ← ThisColumn + 1
                IF Board[ThisRow, ThisColumn] = BLANK
                  THEN
                    BlankFound ← TRUE
                ENDIF
            UNTIL ThisColumn = 7 OR BlankFound = TRUE
        UNTIL ThisRow = 6 OR BlankFound = TRUE
        IF BlankFound = FALSE
          THEN
            OUTPUT "It is a draw"
            GameFinished ← TRUE
        ENDIF
    ENDPROCEDURE
```

Identifier	Data type	Explanation
`BlankFound`	`BOOLEAN`	`FALSE` if no blank location found on the board `TRUE` if a blank location found on the board
`ThisRow`	`INTEGER`	Loop counter for rows
`ThisColumn`	`INTEGER`	Loop counter for columns

Table 13.13 Additional identifiers for the `CheckForFullBoard` procedure

```
    PROCEDURE SwapThisPlayer
        IF ThisPlayer = 'O'
          THEN
            ThisPlayer ← 'X'
          ELSE
            ThisPlayer ← 'O'
        ENDIF
    ENDPROCEDURE
```

We can also use arrays of records. Using the Person record type from Section 13.02, we can declare an array of that type for 100 person records:

```
DECLARE Person : ARRAY[1:100] OF PersonType
```

We can then access an individual's data. For example the first person's name in the array is set as follows:

```
Person[1].Name ← "Fred"
OUTPUT Person[1].Name
```

This is particularly useful when we have several people's data to work with and do not want to use a separate 1D array for each field.

TASK 13.07

1. Declare an array of BookType (see Task 13.02) for 200 books.
2. Set the first book's details to the values given in Task 13.02.

13.06 Text files

Data need to be stored permanently. One approach is to use a file. For example, any data held in an array while your program is executing will be lost when the program stops. You can save the data out to a file and read it back in when your program requires it on subsequent executions.

A text file consists of a sequence of characters formatted into lines. Each line is terminated by an end-of-line marker. The text file is terminated by an end-of-file marker.

TIP

You can check the contents of a text file (or even create a text file required by a program) by using a text editor such as NotePad.

Writing to a text file

Writing to a text file usually means creating a text file.

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE       // open the file for writing
WRITEFILE <filename>, <stringValue>  // write a line of text to the file
CLOSEFILE <filename>               // close file
```

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ          // open file for reading
READFILE <filename>, <stringVariable>   // read a line of text from the file
CLOSEFILE <filename>                 // close file
```

Appending to a text file

Sometimes we may wish to add data to an existing file rather than create a new file. This can be done in Append mode. It adds the new data to the end of the existing file.

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND     // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>              // close file
```

The end-of-file (EOF) marker

If we want to read a file from beginning to end, we can use a conditional loop. Text files contain a special marker at the end of the file that we can test for. Testing for this special end-of-file marker is a standard function in many programming languages. Every time this function is called it will test for this marker. The function will return FALSE if the end of the file is not yet reached and will return TRUE if the end-of-file marker has been reached.

In pseudocode we call this function `EOF()`. We can use the construct `REPEAT...UNTIL EOF()`. If it is possible that the file contains no data, it is better to use the construct `WHILE NOT EOF()`.

For example, the following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
    READFILE "Test.txt", TextString
    OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

TASK 13.08

1. Write pseudocode to save the array data from Task 13.06 to a text file.
2. Write pseudocode to read the values stored in the text file back into the board array.

13.07 Abstract Data Types (ADTs)

An **Abstract Data Type** is a collection of data and a set of associated operations:

- create a new instance of the data structure
- find an element in the data structure
- insert a new element into the data structure
- delete an element from the data structure
- access all elements stored in the data structure in a systematic manner.

The remainder of this chapter describes the following ADTs: stack, queue and linked list. It also demonstrates how they can be implemented from arrays.

In the following ADTs data items are represented as a single character, but this would normally be a set of data, possibly stored as fields in a record.

13.08 Stacks

What are the features of a stack in the real world? To make a stack, we pile items on top of each other. The item that is accessible is the one on top of the stack. If we try to find an item in the stack and take it out, we are likely to cause the pile of items to collapse.

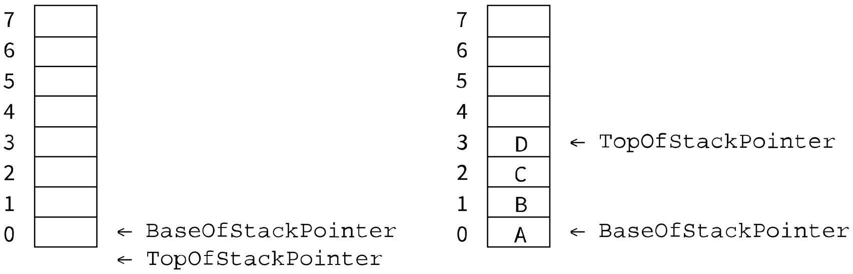


Figure 13.07 An empty stack (left) and a stack with four items pushed (right)

Figure 13.07 shows how we can represent a stack when we have added four items in this order: A, B, C, D. Note that the slots are shown numbered from the bottom as this feels more natural.

The `BaseOfStackPointer` will always point to the first slot in the stack. The `TopOfStackPointer` will point to the last element pushed (added) onto the stack. When an element is popped (removed) from the stack, the `TopOfStackPointer` will decrease to point to the element now at the top of the stack. When the stack is empty, `TopOfStackPointer` will have the value –1.

To implement this stack using a 1D array, we write:

```
DECLARE Stack:ARRAY[0 : 7] OF CHAR
```

TASK 13.09

1. Draw a diagram to show the contents of the stack shown in Figure 13.07 after “E” has been pushed onto the stack.
2. Draw a diagram to show the contents of the stack shown in Figure 13.07 after one item has been popped off the stack.

13.09 Queues

What are the features of a queue in the real world? When people form a queue, they join the queue at the end. People leave the queue from the front of the queue. If it is an orderly queue, no-one pushes in between and people don't leave the queue from any other position.

Figure 13.08 shows how we can represent a queue when five items have joined the queue in this order: A, B, C, D, E.

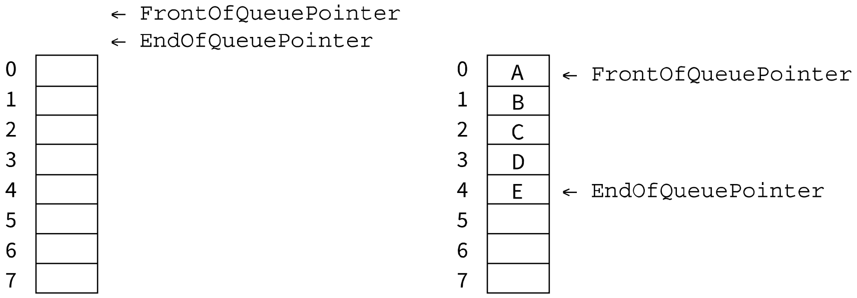

← FrontOfQueuePointer
← EndOfQueuePointer
0
1
2
3
4
5
6
7
0 A ← FrontOfQueuePointer
1 B
2 C
3 D
4 E ← EndOfQueuePointer
5
6
7

Figure 13.08 An empty queue (left) and a queue after 5 items have joined (right)

To implement a queue using an array, we can assume that the front of the queue is at position 0. When the queue is empty, the `EndOfQueuePointer` will have the value –1. When one value joins the queue, the `EndOfQueuePointer` will be incremented before adding the value to the array element where the pointer is pointing to. When the item at the front of the queue leaves, we need to move all the other items one slot forward and adjust `EndOfQueuePointer`.

TASK 13.10

1 Draw a diagram to show the contents of the queue after "F" has joined the non-empty queue shown in Figure 13.08.

2 Draw a diagram to show the contents of the queue after one item has left the non-empty queue shown in Figure 13.08.

This method involves a lot of moving of data. A more efficient way to make use of the slots is the concept of a 'circular' queue. Pointers show where the front and end of the queue are. Eventually the queue will 'wrap around' to the beginning. Figure 13.09 shows a circular queue after 11 items have joined and five items have left the queue.

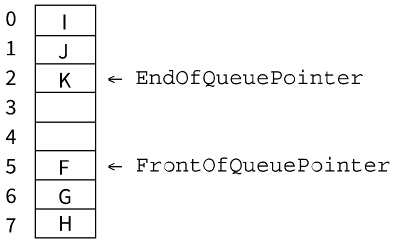

0 I
1 J
2 K ← EndOfQueuePointer
3
4
5 F ← FrontOfQueuePointer
6 G
7 H

Figure 13.09 A circular queue

13.10 Linked lists

In Section 13.03 we used an array as a linear list. In a linear list, the list items are stored in consecutive locations. This is not always appropriate. Another method is to store an individual list item in whatever location is available and link the individual item into an ordered sequence using pointers.

An element of a linked list is called a node. A **node** can consist of several data items and a **pointer**, which is a variable that stores the address of the node it points to.

A pointer that does not point at anything is called a **null pointer.** It is usually represented by ∅. A variable that stores the address of the first element is called a **start pointer.**

In Figure 13.10, the data value in the node box represents the key field of that node. There are likely to be many data items associated with each node. The arrows represent the pointers. It does not show at which address a node is stored, so the diagram does not give the value of the pointer, only where it conceptually links to.

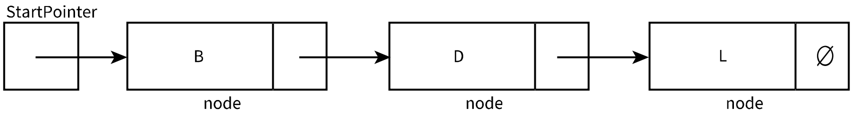


Figure 13.10 Conceptual diagram of a linked list

Figure 13.11 shows how a new node, A, is inserted at the beginning of the list. The content of `StartPointer` is copied into the new node's pointer field and `StartPointer` is set to point to the new node, A.

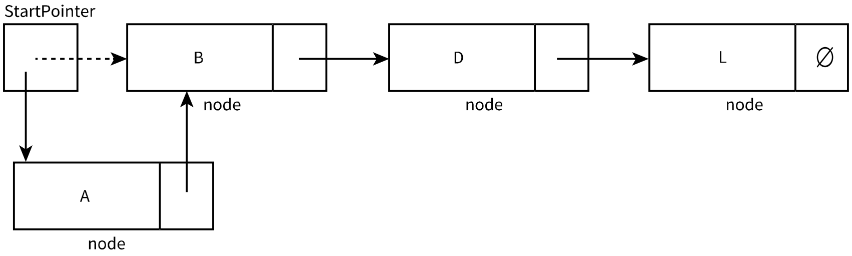


Figure 13.11 Conceptual diagram of adding a new node to the beginning of a linked list

In Figure 13.12, a new node, P, is inserted at the end of the list. The pointer field of node L points to the new node, P. The pointer field of the new node, P, contains the null pointer.

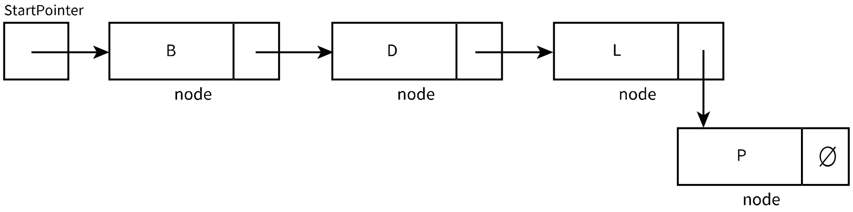


Figure 13.12 Conceptual diagram of adding a new node to the end of a linked list

To delete the first node in the list (see Figure 13.13), we copy the pointer field of the node to be deleted into `StartPointer`.

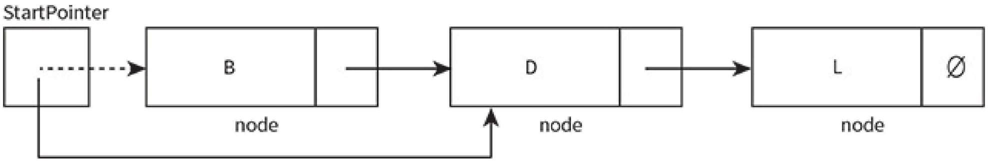


Figure 13.13 Deleting the first node in a linked list

To delete the last node in the list (see Figure 13.14), we set the pointer field for the previous node to the null pointer.

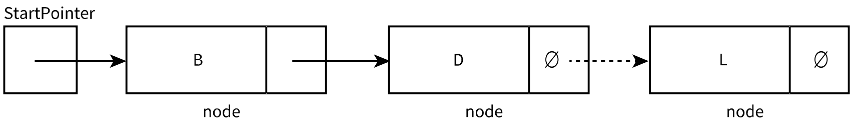


Figure 13.14 Conceptual diagram of deleting the last node of a linked list

Sometimes the nodes are linked together in order of key field value to produce an ordered linked list. This means a new node may need to be inserted or deleted from between two existing nodes.

To insert a new node, C, between existing nodes, B and D (see Figure 13.15), we copy the pointer field of node B into the pointer field of the new node, C. We change the pointer field of node B to point to the new node, C.

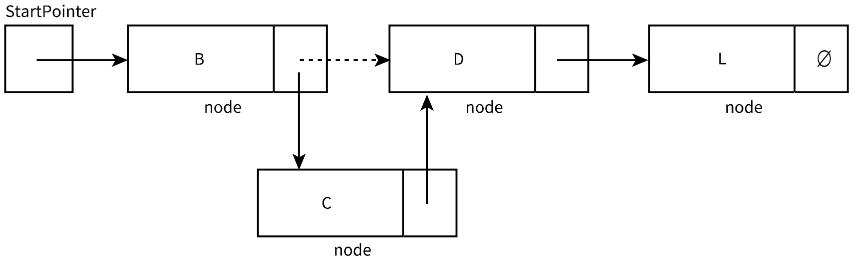


Figure 13.15 Conceptual diagram of adding a new node into a linked list

To delete a node, D, within the list (see Figure 13.16), we copy the pointer field of the node to be deleted, D, into the pointer field of node B.

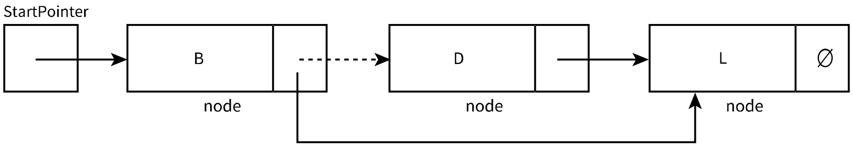


Figure 13.16 Conceptual diagram of deleting a node within a linked list

Remember that, in real applications, the data would consist of much more than a key field and one data item. This is why linked lists are preferable to linear lists. When list elements need reordering, only pointers need changing in a linked list. In a linear list, all data items would need to be moved.

Using linked lists saves time, however we need more storage space for the pointer fields.

To implement a linked list using arrays, we can use a 1D array to store the data and a 1D array to store the pointer. Reading the array values across at the same index, one row represents a node.

A value is added to the next free element of the `Data` array and pointers are adjusted to incorporate the node in the correct position within the linked list.

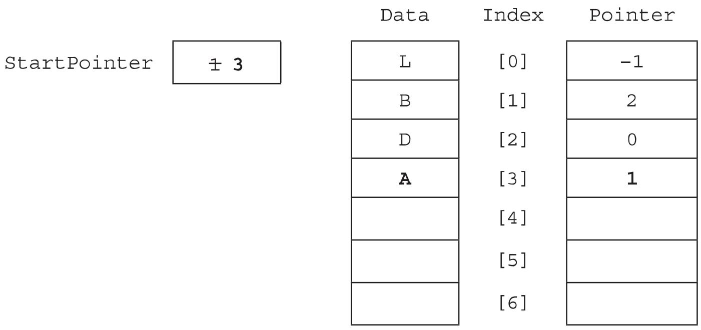
Figure 13.17 shows how two arrays can be used to implement the linked list from Figure 13.10.

`StartPointer` `1`

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.17 Using arrays to implement the linked list shown in Figure 13.10

Figure 13.18 shows how a new node is added to the beginning of a linked list implemented using arrays. Note that the value “A” is added at index 3 but the start pointer is adjusted to make this the new first element of the list.

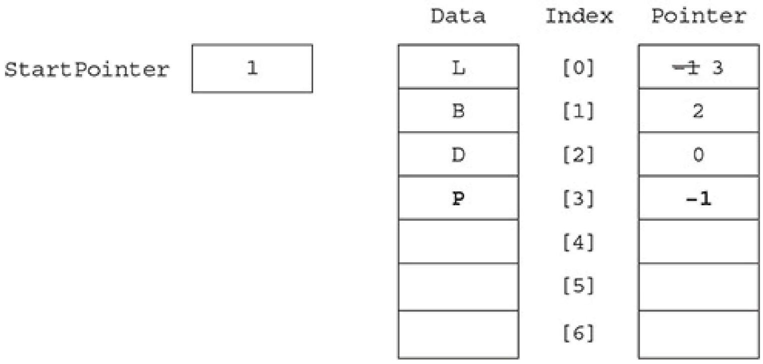


StartPointer ~~1~~ 3

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
A	[3]	**1**
	[4]	
	[5]	
	[6]	

Figure 13.18 Adding a new node to the beginning of a linked list

Figure 13.19 shows how a new node is added to the end of a linked list implemented using arrays. Note that the value “P” is added at index 3. The node that previously contained the null pointer (at index 0) has its pointer adjusted to point to the new node.



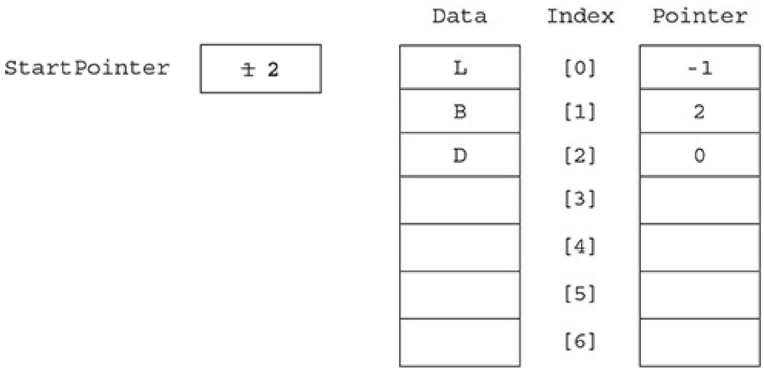
StartPointer 1

Data	Index	Pointer
L	[0]	~~-1~~ 3
B	[1]	2
D	[2]	0
P	[3]	**-1**
	[4]	
	[5]	
	[6]	

Figure 13.19 Adding a new node to the end of a linked list implemented using arrays

When deleting a node, only pointers need to be adjusted. The old data can remain in the array, but it will no longer be accessible as no pointer will point to it.

Figure 13.20 shows how the start pointer is adjusted to effectively delete the first element of the linked list. Note that the start pointer now contains the pointer value of the deleted node.

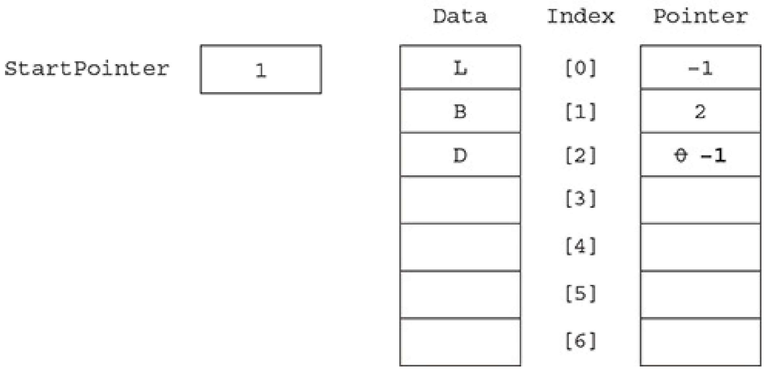


StartPointer ~~1~~ 2

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.20 Deleting the first node in a linked list implemented using arrays

Figure 13.21 shows how the pointer value of the penultimate node of the linked list is changed to the null pointer.

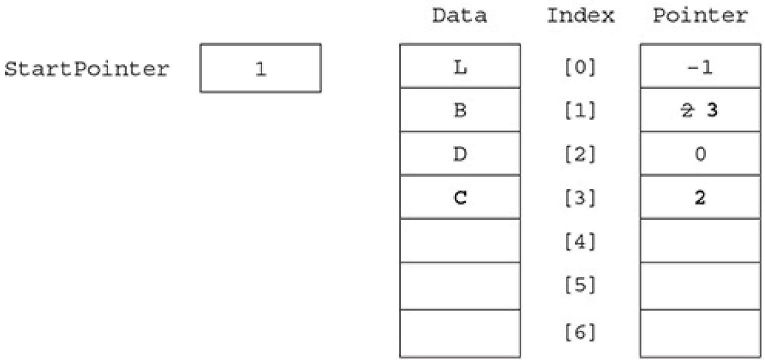


StartPointer: 1

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	~~0~~ -1
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.21 Deleting the last node of a linked list implemented using arrays

When adding a node that needs to be inserted into the list, the data is added to any free element of the Data array. The pointer of the new node is set to point to the index of the node that comes after the insertion point. Note that this is the value of the pointer of the node preceding the insertion point. The pointer of the node preceding the insertion point is set to point to the new node.

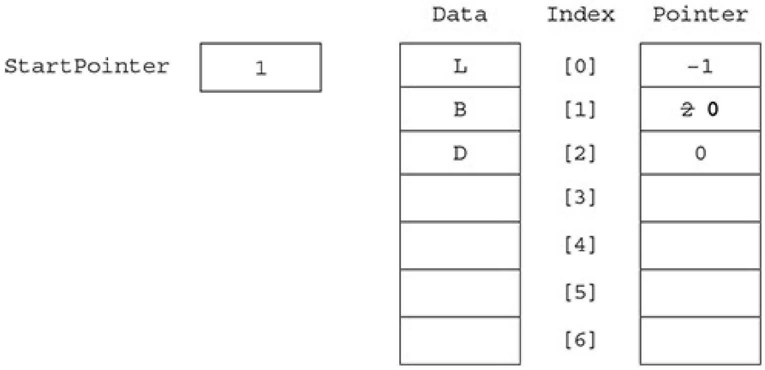


StartPointer: 1

Data	Index	Pointer
L	[0]	-1
B	[1]	~~2~~ 3
D	[2]	0
C	[3]	2
	[4]	
	[5]	
	[6]	

Figure 13.22 Adding a new node into a linked list implemented using arrays

Again, when deleting a node, only pointers need to be adjusted. Figure 13.23 shows how the pointer of the node to be deleted is copied into the pointer of the preceding node.



StartPointer: 1

Data	Index	Pointer
L	[0]	-1
B	[1]	~~2~~ 0
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.23 Deleting a node within a linked list implemented using arrays

Unused nodes need to be easy to find. A suitable technique is to link the unused nodes to form another linked list: the free list. Figure 13.24 shows our linked list and its free list.

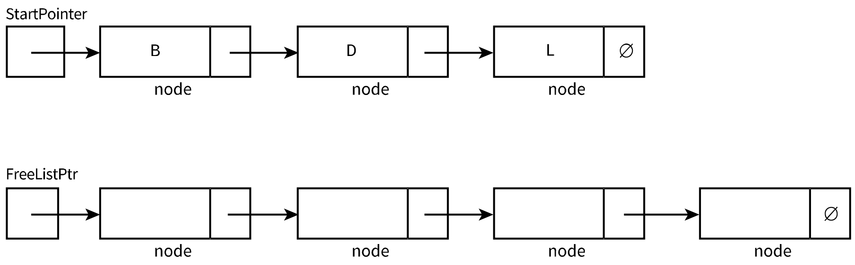
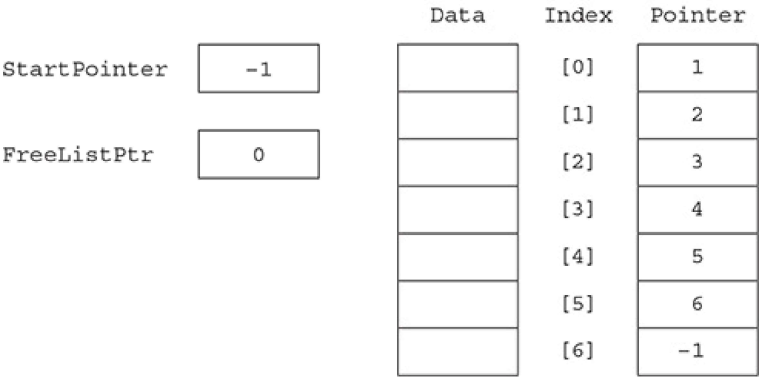


Figure 13.24 Conceptual diagram of a linked list and a free list

When an array of nodes is first initialised to work as a linked list, the linked list will be empty. So the start pointer will be the null pointer. All nodes need to be linked to form the free list. Figure 13.25 shows an example of an implementation of a linked list before any data is inserted into it.



StartPointer -1

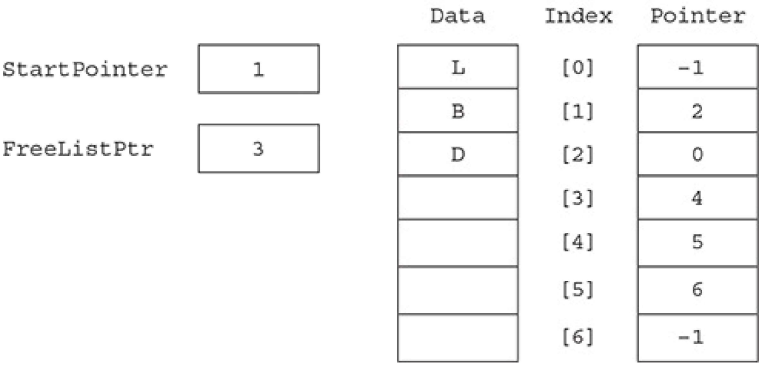
FreeListPtr 0

Data	Index	Pointer
	[0]	1
	[1]	2
	[2]	3
	[3]	4
	[4]	5
	[5]	6
	[6]	-1

Figure 13.25 A linked list and a free list implemented using arrays

Assume “L”, “B” and “D” were added to the linked list and to be kept in alphabetical order.

Figure 13.26 shows how the values are stored in the `Data` array and the pointers of the linked list and free list adjusted.



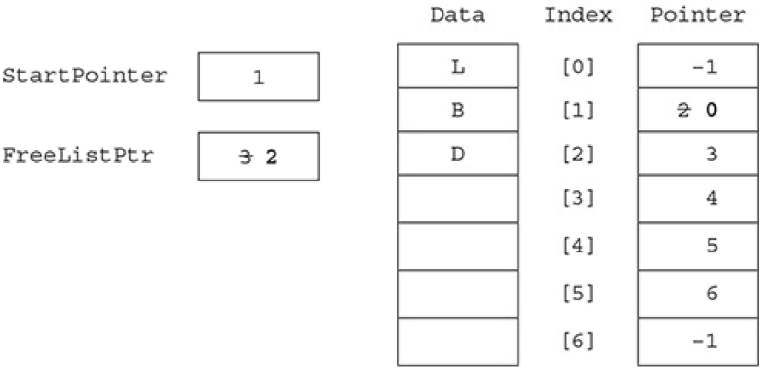
StartPointer 1

FreeListPtr 3

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
	[3]	4
	[4]	5
	[5]	6
	[6]	-1

Figure 13.26 Linked list and free list implemented using arrays

If the node containing “B” is to be deleted, the array element of that node needs to be linked back into the free list. Figure 13.27 shows how this is done by adding the node to the front of the free list.



StartPointer 1

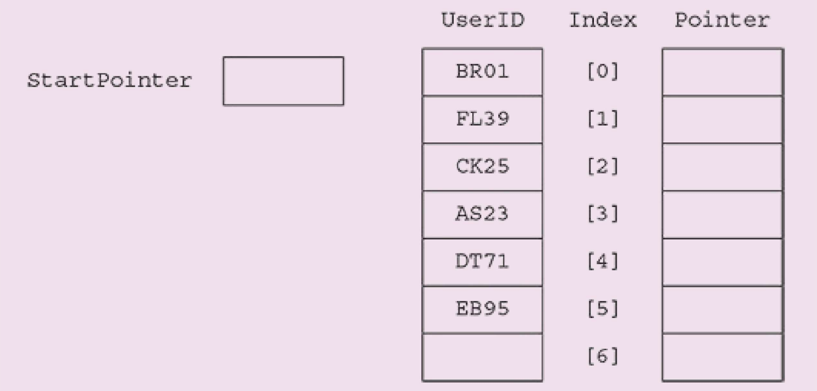
FreeListPtr ~~3~~ 2

Data	Index	Pointer
L	[0]	-1
B	[1]	~~2~~ 0
D	[2]	3
	[3]	4
	[4]	5
	[5]	6
	[6]	-1

Figure 13.27 Linked list and free list implemented using arrays

TASK 13.11

A linked list is to be set up using the values in the `UserID` array shown below.



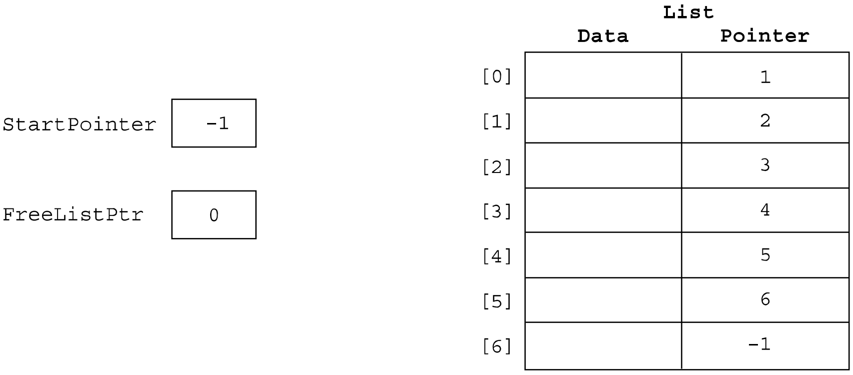
StartPointer ☐

UserID	Index	Pointer
BR01	[0]	
FL39	[1]	
CK25	[2]	
AS23	[3]	
DT71	[4]	
EB95	[5]	
	[6]	

Without moving any of the contents of the `UserID` array, insert the pointer values so that the linked list is in alphabetical order.

In Section 13.02 we looked at the user-defined record type. We grouped together related data items into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

We can store the linked list in an array of records. One record represents a node and consists of the data and a pointer (see Figure 13.28).



StartPointer: -1

FreeListPtr: 0

	List	
	Data	**Pointer**
[0]		1
[1]		2
[2]		3
[3]		4
[4]		5
[5]		6
[6]		-1

Figure 13.28 A linked list before any nodes are used

TIP

A stack can be implemented from a linked list. The start pointer is seen as the top of stack pointer and a data item is only added to the start of the linked list and a node is only removed from the start of the linked list.

TIP

A queue can be implemented from a linked list. The start pointer is seen as the front of the queue. Data items are always added to the end of the linked list and items are always removed from the start of the linked list.

Reflection Point:

What is the difference between standard data types, ADTs and user-defined data types?

Summary

- Standard data types are INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE.
- A record structure holds a set of data of different data types under one identifier.
- Use the dot notation to address fields of a record.

- Arrays have dimensions with upper and lower bounds.
- Individual array elements are accessed using an index (1D arrays) or two indexes (2D arrays).
- A bubble sort algorithm compares pairs of values in a linear list and swaps them if required.
- A linear search checks each value in turn for a required value.
- Text files can be written to and read from and store data between program runs.
- Stacks, queues and linked lists are examples of Abstract Data Types (ADTs) and can be implemented using arrays.

Exam-style Questions

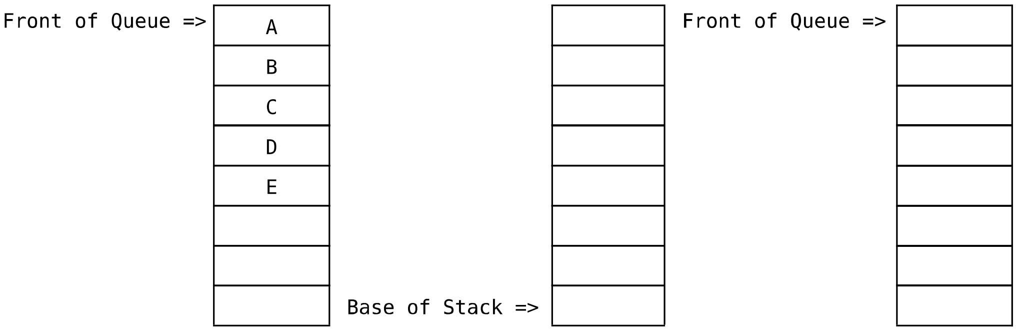
1 Complete the following variable identifier table:

Variable	**Example value**	**Data type**
`ColourCode`	***"034AB45"***	
`ProductionDate`	***2018/03/31***	
`Weight`	***67.45***	
`NumberInStock`	***98***	
`SizeCode`	***'X'***	
`Completed`	***FALSE***	

[3]

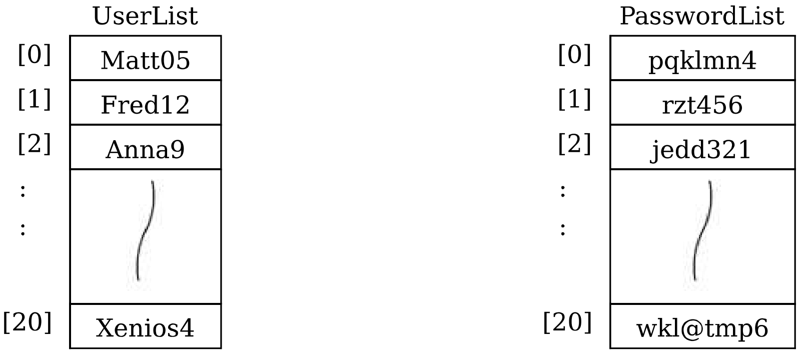
2 A stack and a queue are used to reverse the order of a set of values.

Complete the diagram:


Front of Queue =>
A
B
C
D
E
Base of Stack =>
Front of Queue =>

[2]

3 Alicia uses two 1D arrays, `UserList` and `PasswordList`. For twenty users, she stores each user ID in `UserList` and the corresponding password in `PasswordList`. For example, the person with user ID `Fred12` has password `rzt456`.


UserList
[0] Matt05
[1] Fred12
[2] Anna9
:
:
[20] Xenios4
PasswordList
[0] pqklmn4
[1] rzt456
[2] jedd321
:
:
[20] wkl@tmp6

Alicia wants to write an algorithm to check whether a user ID and password, entered by a user, are correct. She designs the algorithm to search `UserList` for the user ID. If the user ID is found, the password stored in `PasswordList` is to be compared to the entered password. If the passwords match, the login is successful. In all other cases, login is unsuccessful.

a Complete the identifier table. [8]

Identifier	**Data type**	**Explanation**
`UserList`		1D array to store user IDs
………………		1D array to store passwords
`MaxIndex`		Upper bound of the array
`MyUserID`		User ID entered to login

`MyPassword`		....
`UserIdFound`		FALSE if user ID not found in UserList TRUE if ....
`LoginOK`		FALSE if .... TRUE if ....
`Index`		Pointer to current array element

b Complete the pseudocode for Alicia’s algorithm:

```
MaxIndex ← 20
INPUT MyUserID
INPUT MyPassword
UserIdFound ← FALSE
LoginOK ← ...............
Index ← −1
REPEAT
    INDEX ← ...............
    IF UserList[...............] = ...............
      THEN
        UserIdFound ← TRUE
    ENDIF
UNTIL ............... OR ...............
IF UserIdFound = TRUE
  THEN
    IF PasswordList[...............] = ...............
      THEN
        LoginOK ← TRUE
    ...............
ENDIF
IF ...............
  THEN
    OUTPUT "Login successful"
  ELSE
    OUTPUT "User ID and/or password incorrect"
ENDIF
```

[10]

c **i** Instead of using two 1D arrays, Alicia could have used an array of records.

Write pseudocode to declare the record structure `UserRecord`. [2]

ii Write pseudocode to declare the `User` array. [2]

Chapter 14:

Programming and data representation

Learning objectives

By the end of this chapter you should be able to:

- write a program in a high-level language (Python, Visual Basic console mode, Java)
- implement and write pseudocode from a given design presented as either a program flowchart or structured English
- write pseudocode and program statements for:
    - the declaration of variables and constants
    - the assignment of values to variables and constants
    - expressions involving any of the arithmetic or logical operators
    - input from the keyboard and output to the console
- use a subset of the built-in functions and library routines supported by the chosen programming language, including those used for string/character manipulation and random number generator
- use an ‘IF’ structure including the ‘ELSE’ clause and nested IF statements
- use a ‘CASE’ structure
- use a loop (‘count controlled’, ‘post-condition’, ‘pre-condition’)
- justify why one loop structure may be better suited to a problem than the others
- define and use a procedure and explain where in the construction of an algorithm it is appropriate to use a procedure
- use parameters
- show understanding of passing parameters by reference and by value
- define and use a function and explain where in the construction of an algorithm it is appropriate to use a function
- show understanding that a function is used in an expression (the return value replaces the call)
- use the terminology associated with procedures and functions: procedure/function header, procedure/function interface, parameter, argument, return value
- write efficient code.

14.01 Programming languages

Chapters 12 and 13 introduced the concept of solving a problem and representing a solution using a flowchart or pseudocode. We expressed our solutions using the basic constructs: assignment, sequence, selection, iteration, input and output.

To write a computer program, we need to know the syntax (the correct structure of statements) of these basic constructs in our chosen programming language. This chapter introduces syntax for Python, Visual Basic console mode and Java.

Note that for convenience and easy reference, definitive pseudocode syntax is repeated in this chapter at the appropriate points.

You only need learn to program in one of the three languages covered in this book. Programming language is only examined at A Level but it is important to start learning it from AS Level to ensure you are well-prepared. For the AS Level exams you should use pseudocode rather than programming language.

TIP

The only way of knowing whether the algorithm you have designed is a suitable solution to the problem you are trying to solve, is to implement your pseudocode in your chosen programming language and test the program by running it.

Python

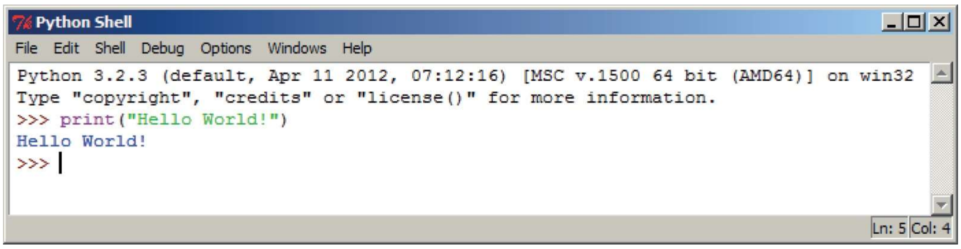
Python was conceived by Guido van Rossum in the late 1980s. Python 2.0 was released in 2000 and Python 3.0 in 2008. Python is a multi-paradigm programming language, meaning that it fully supports both object-oriented programming and structured programming. Many other paradigms, including logic programming, are supported using extensions. These paradigms are covered in Chapters 26, 27 and 29.

The Python programs in this book have been prepared using Python 3 (see Python for a free download) and Python's Integrated DeveLopment Environment (IDLE).

Key characteristics of Python include the following.

- Every statement must be on a separate line.
- Indentation is significant. This is known as the 'off-side rule'.
- Keywords are written in lower case.
- Python is case sensitive: the identifier `Number1` is seen as different from `number1` or `NUMBER1`.
- Everything in Python is an object (see Chapter 27).
- Code makes extensive use of a concept called 'slicing' (see Section 14.07).
- Programs are interpreted (see Chapter 8, Section 8.05 for information on interpreted and compiled programs).

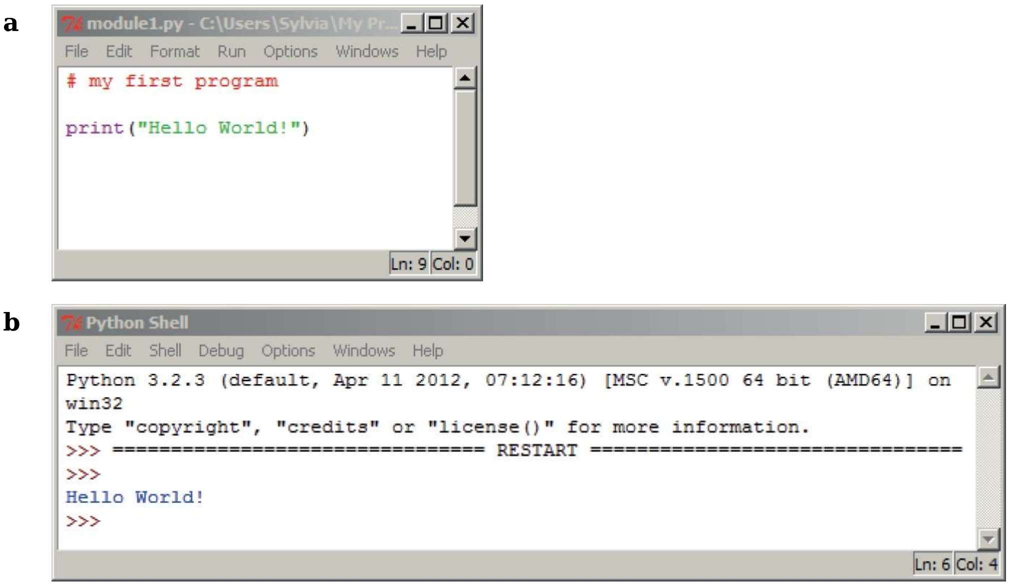
You can type a statement into the Python Shell and the Python interpreter will run it immediately (see Figure 14.01).



```
Python 3.2.3 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>> 
```

Figure 14.01 Running a statement in the Python shell

You can also type program code into a Python editor (such as IDLE), save it with a .py extension and then run the program code from the Run menu in the editor window (see Figure 14.02).


a
module1.py - C:\Users\Sylvia\My Pr...
File Edit Format Run Options Windows Help
Ln: 9 Col: 0
b
Python Shell
File Edit Shell Debug Options Windows Help
Ln: 6 Col: 4

```
# my first program

print("Hello World!")
```

```
Python 3.2.3 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello World!
>>>
```

Figure 14.02 (a) A saved program in the Python editor window and (b) running in the Python shell

Visual Basic Console Mode (VB.NET)

VB.NET is a multi-paradigm, high-level programming language, implemented on the .NET Framework. Microsoft launched VB.NET in 2002 as the successor to its original Visual Basic language. Microsoft’s integrated development environment (IDE) for developing in VB.NET is Visual Studio. Visual Studio Express and Visual Studio Community are freeware.

The Visual Basic programs in this book have been prepared using Microsoft Visual Basic 2010 Express Console Application. (Free download available from Visual Studio Express)

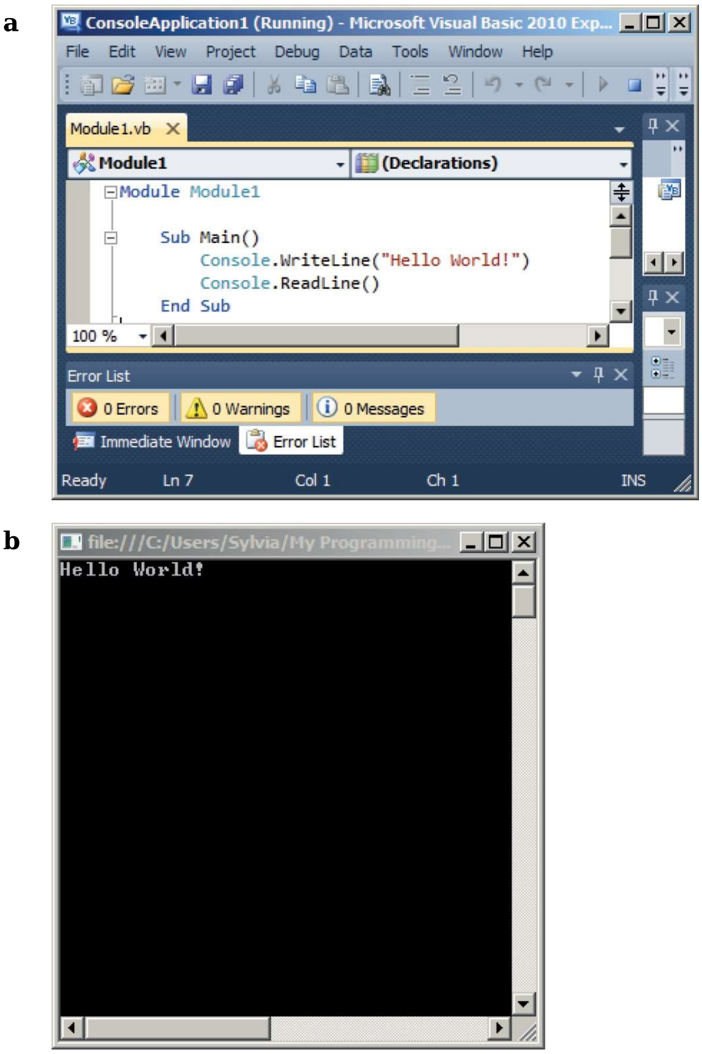
Key characteristics of VB.NET include the following.

- Every statement should be on a separate line. Statements can be typed on the same line with a colon (:) as a separator. However, this is not recommended.
- Indentation is good practice.
- VB.NET is not case sensitive. Modern VB.NET editors will automatically copy the case from the first definition of an identifier.
- The convention is to use CamelCaps (also known as PascalCaps) for identifiers and keywords.
- Programs need to be compiled (see Chapter 8, Section 8.05 for information on interpreted and compiled programs).

You type your program code into the Integrated Development Environment (IDE) as shown in Figure 14.03 (a), save the program code and then click on the Run button ▶. This starts the compiler. If there are no syntax errors the compiled program will then run. Output will be shown in a separate console window (see Figure 14.03 (b)).

Note that the console window shuts when the program has finished execution. To keep the console window open so you can see the output (see Figure 14.03), the last statement of your program should be

```
Console.ReadLine()
```


a
ConsoleApplication1 (Running) - Microsoft Visual Basic 2010 Exp...
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
(Declarations)
100 %
Error List
0 Errors
0 Warnings
0 Messages
Immediate Window
Error List
Ready Ln 7 Col 1 Ch 1 INS
b
file:///C:/Users/Sylvia/My Programming...

```
Module Module1

    Sub Main()
        Console.WriteLine("Hello World!")
        Console.ReadLine()
    End Sub
```

```
Hello World!
```

Figure 14.03 (a) A saved program in the VB.NET editor and (b) running in the program execution (console) window

Java

Java was originally developed by James Gosling at Sun Microsystems (now owned by Oracle) and released in 1995. The Java Runtime Environment (JRE) is intended for end users, and the Java Development Kit (JDK) is intended for software developers and includes development tools such as the Java compiler and a debugger.

Java was intended to be platform independent. The Java programs in this book have been prepared using NetBeans 8.2. However, any text editor can be used to write Java source code.

Key characteristics of Java include the following.

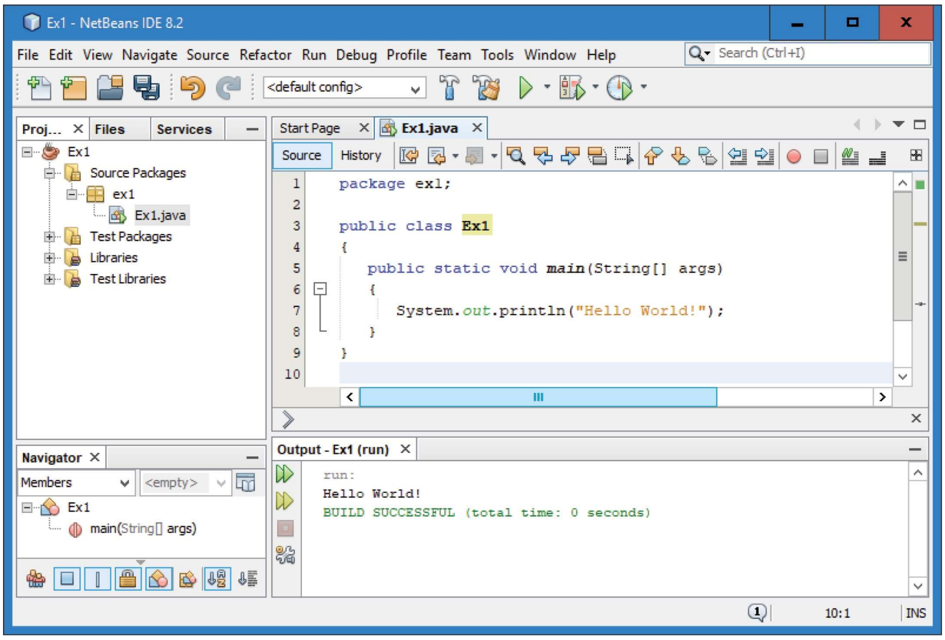
- Every statement ends with a semicolon (`;`). More than one statement can go on a single line, but this is not recommended.
- Indentation is good practice.
- Java is case sensitive.
- The convention is to use camelCaps for identifiers, lower case for keywords and capitalised identifiers for classes.
- A compound statement consists of a sequence of statements enclosed between braces `{ }`.
- Whenever Java syntax requires a statement, a compound statement can be used.
- Programs need to be compiled (see Chapter 8, Section 8.05 for information on interpreted and compiled programs) into bytecode and then run using the Java Virtual Machine.

Java was designed almost exclusively as an object-oriented language. All code is written inside classes. Only the simple data types (such as integer, real) are not objects. Strings are objects.

Source files must be named after the public class they contain, appending the suffix .java, for example, Ex1.java. It must first be compiled into bytecode, using a Java compiler, producing a file named Ex1.class. Only then can it be executed.

The method name “main” is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program.

You type your program statements into the Integrated Development Environment (IDE) as shown in Figure 14.04, save the program code and then click on the Run button (▶). This starts the compiler. If there are no syntax errors the compiled program code will then run. Output will be shown in the Output window (see Figure 14.04).


Ex1 - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Proj... Files Services
Ex1
Source Packages
ex1
Ex1.java
Test Packages
Libraries
Test Libraries
Start Page
Ex1.java
Source History
Navigator
Members
<empty>
Ex1
main(String[] args)
Output - Ex1 (run)
10:1
INS

```
package ex1;

public class Ex1
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

```
run:
Hello World!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 14.04 A Java program in the NetBeans editor and running in the Output window

14.02 Programming basics

Declaration of variables

Most programming languages require you to declare the type of data to be stored in a variable, so the correct amount of memory space can be reserved by the compiler. A variable declared to store a whole number (integer) cannot then be used to store alphanumeric characters (strings), or the other way around. VB.NET and Java require variables to be declared before they are used.

Python handles variables differently to most programming languages. It tags values. This is why Python does not have variable declarations.

TIP

It is good programming practice to include a comment about the variables you are planning to use and the type of data you will store in them.

In pseudocode, variable declarations are written as:

```
DECLARE <identifier> : <dataType>
```

For example, you may declare the following variables:

```
DECLARE Number1 : INTEGER // this declares Number1 to store a whole number
DECLARE YourName : STRING // this declares YourName to store a
                          // sequence of characters
DECLARE N1, N2, N3 : INTEGER  // declares 3 integer variables
DECLARE Name1, Name2 : STRING // declares 2 string variables
```

Syntax definitions

The syntax of variable declarations in language code is as follows:

Python	Python does not have variable declarations
VB.NET	`Dim` <identifier>[, <identifier>] `As <dataType>` Each line of declarations must start with the keyword `Dim`.
Java	`<datatype> <identifier>[, <identifier>];`

Code examples

Python	`# Number1 of type Integer`
	`# YourName of type String`
	`# N1, N2, N3 of type integer;`
	`# Name1, Name2 of type string;`
		There are no declarations, but comments should be made at the beginning of a module (see the section about comments at the end of Section 14.02).

VB.NET	`Dim Number1 As Integer`
	`Dim YourName As Integer`
	`Dim N1, N2, N3 As Integer`
	`Dim Name1, Name2 As String`
		You can group more than one variable of the same type on the same line.

Java	`int number1;`
	`String yourName;`
	`int n1, n2, n3;`
	`String name1, name2;`
		You can group more than one variable of the same type on the same line.

Declaration and assignment of constants

Sometimes we use a value in a solution that never changes, for example, the value of the mathematical constant pi (π). Instead of using the actual value in program statements, it is good practice and helps readability, if we give a constant value a name and declare it at the beginning of the program.

In pseudocode, constant declarations are written as:

```
CONSTANT <identifier> = <value>
```

For example:

```
CONSTANT Pi = 3.14
```

Syntax definitions

Python	**`<identifier> = <value>`**
VB.NET	`Const` **`<identifier> = <value>`** Each line of declarations must start with the keyword `Const`.

Java	`static final <datatype> <identifier> = <value>;`
	Each line of constant declarations must start with the keywords
	`static final`

Code examples

Python	`PI = 3.14`	Python convention is to write constant identifiers using uppercase only. The values can be changed, although you should treat constants as not changeable.
VB.NET	`Const Pi = 3.14`	The value of a constant in VB.NET cannot be altered within the program.
Java	`static final double PI = 3.14;`	The value of a constant in Java cannot be altered within the program.

Assignment of variables

Once we have declared a variable, we can assign a value to it (See Chapter 12, Section 12.05).

In pseudocode, assignment statements are written as:

```
<identifier> ← <expression>
```

For example:

```
A ← 34
B ← B + 1
```

Syntax definitions and code examples

Python	`<identifier> = <expression>`	`A = 34` `B = B + 1`	The assignment operator is =
VB.NET	`<identifier> = <expression>`	`A = 34` `B = B + 1`	The assignment operator is =
Java	`<identifier> = <expression>;`	`A = 34;` `B = B + 1;`	The assignment operator is =

VB.NET allows you to initialise a variable as part of the declaration statement, for example:

```
Dim Number1 As Integer = 0
```

Java allows you to initialise a variable as part of the declaration statement, for example:

```
int number1 = 0;
```

VB.NET and Python allow increment statements such as B = B + 1 to be written as B += 1.

Java allows increment statements such as `b = b + 1` to be written as `b++;`

Arithmetic operators

Assignments don't just give initial values to variables. We also use an assignment when we need to store the result of a calculation. The arithmetic operators used for calculations are shown in Table 14.01.

Operation	Pseudocode	Python	VB.NET	Java
Addition	`+`	`+`	`+`	`+`
Subtraction	`-`	`-`	`-`	`-`
Multiplication	`*`	`*`	`*`	`*`
Division	/	/	/	/ (when dividing float or double types)
Exponent	^	**	^	No operator available, only method: `Math.pow(n,e)`
Integer division	`DIV`	`//`	`\`	/ (when dividing integer types)
Modulus	`MOD`	`%`	`Mod`	`%`

Table 14.01 Arithmetic operators

TIP

The result of integer division is the whole number part of the division. For example, 7 DIV 2 gives 3.

TIP

The result of the modulus operation is the remainder of a division. For example, 7 MOD 2 gives 1.

When more than one operator appears in an expression, the order of evaluation depends on the mathematical **rules of precedence**: parentheses, exponentiation, multiplication, division, addition, subtraction.

Question 14.01

Evaluate each of the following expressions:

4 * 3 – 3 ^ 2

(4 * 3 – 3) ^ 2

4 * (3 – 3) ^ 2

4 * (3 – 3 ^ 2)

Outputting information to the screen

In pseudocode, output statements are written as:

```
OUTPUT <string>
OUTPUT <identifier(s)>
```

When outputting text and data to the console screen, we can list a mixture of output strings and variable values in the print list.

Syntax definitions

Python	`print(<printlist>)` `print(<printlist>, end ='')`	Print list items are separated by commas (,). To avoid moving onto the next line after the output, use `end ='')`
VB.NET	`Console.WriteLine(<printlist>)` `Console.Write(<printlist>)`	Print list items are joined using &. `Console.WriteLine` will move onto the next line after the output; `Console.Write` will remain on the same line.

Java	`System.out.print(<printlist>);` `System.out.println(<printlist>);`	Print list items are joined using +. `System.out.println` will move onto the next line after the output; `System.out.print` will remain on the same line.

In the examples below, the print list consists of four separate items:

“Hello ” and “. Your number is ” are strings and

YourName and Number1 are variables, for which we print the value.

In pseudocode, we can indicate whether a new line should be output at the end by a comment at the end of the statement.

```
OUTPUT "Hello ", YourName, ". Your number is ", Number1 // newline
OUTPUT "Hello " // no new line
```

Code examples

Python

```
print("Hello ", YourName,
   ". Your number is ", Number1)
print("Hello ", end= '')
```

VB.NET

```
Console.WriteLine("Hello " & YourName &
   ". Your number is " & Number1)
Console.Write("Hello")
```

Java

```
System.out.println("Hello " + yourName +
   ". Your number is " + number1);
System.out.print("Hello");
```

In the code examples above you can see how output statements can be spread over more than one line when they are very long. You must break the line between two print list items. You cannot break in the middle of a string, unless you make the string into two separate strings.

In Python and VB.NET you can also use the placeholder method for output: the variables to be printed are represented by sequential numbers in { } in the message string and the variables are listed in the correct order after the string, separated by commas:

Python	`print ("Hello {0}. Your number is {1}".format(YourName, Number1))`
VB.NET	`Console.WriteLine("Hello {0}. Your number is {1}", YourName, Number1)`

Getting input from the user

When coding an input statement, it is good practice to prompt the user as to what they are meant to enter. For example, consider the pseudocode statement:

```
  INPUT "Enter a number: " A
```

Note the space between the colon and the closing quote. This is significant. It gives a space before the user types their input.

Code examples

Python	`A = input("Enter a number: ")`	The prompt is provided as a parameter to the input function. Single quotes are also accepted. All input is taken to be a string; if you want to use the input as a number the input string has to be converted using a function (see Section 14.07).
VB.NET	`Console.Write("Enter a number: ")` `A = Console.ReadLine()`	The prompt has to be supplied as an output statement separately.

Java	`import java.util.Scanner;`
	`Scanner console = new Scanner(System.in);`
	`System.out.print("Enter a number: ");`
	`a = console.next();`
		The Scanner class has to be imported from the Java library first and a scanner object has to be created before it can be used to read an input string.
		The prompt has to be supplied as an output statement separately.

Comments

It is good programming practice to add comments to explain code where necessary.

Code examples

Python

```
# this is a comment
# this is another comment
```

VB.NET

```
' this is a comment
' this is another comment
```

Java

```
// this is a comment
// this is another comment
/* this is a multi-line
   comment
*/
```

TASK 14.01

Use the IDE of your chosen programming language (in future just referred to as 'your language'). Type the program statements equivalent to the following pseudocode (you may need to declare the variable `YourName` first):

```
INPUT "What is your name? " YourName
OUTPUT "Have a nice day ", YourName
```

Save your program as Example1 and then run it. Is the output as you expected?

14.03 Data types

Every programming language has built-in data types. Table 14.02 gives a subset of those available. The number of bytes of memory allocated to a variable of the given type is given in brackets for VB.NET and Java.

Description of data	Pseudocode	Python	VB.NET	Java
Whole signed numbers	`INTEGER`	`int`	`Integer` (4 bytes)	`int` (4 bytes)
Signed numbers with a decimal point	`REAL`	`float`	`Single` (4 bytes) `Double` (8 bytes)	`float` (4 bytes) `double` (8 bytes)
A single character	`CHAR` Use single (`'`) quotation marks to delimit a character	Not available	`Char` (2 bytes – Unicode)	`char` (2 bytes – Unicode)
A sequence of characters (a string)	`STRING` Use double (`"`) quotation marks to delimit a string.	`str` (stored as ASCII but Unicode strings are also available) Use single (`'`), double (`"`) or triple (`'''` or `"""`) quotation marks to delimit a string.	`String` (2 bytes per character) Use double (`"`) quotation marks to delimit a string.	`String` (2 bytes per character) Use double (`"`) quotation marks to delimit a string.

Logical values: True (represented as 1) and False (represented as 0)	`BOOLEAN`	`bool`
		possible values:
		`True`
		`False`
			`Boolean` (2 bytes)
			possible values:
			`True`
			`False`
				`Boolean`
				possible values:
				`true`
				`false`

Table 14.02 Simple data types

In Python, a single character is represented as a string of length 1.

In VB.NET, each character in a string requires two bytes of memory and each character is represented in memory as Unicode (in which, the values from 1 to 127 correspond to ASCII).

Date has various internal representations but is output in conventional format.

Description of data	Pseudocode	Python	VB.NET	Java
Date value	`DATE`	Use the datetime class	`Date` (8 bytes)	Date is a class in Java. To make use of it use: `import java.util.Date;`

Table 14.03 The Date data types

In Python and Java, date is not available as a built-in data type. Date is provided as a class (see Table 14.03).

VB.NET stores dates and times from 1.1.0001 (0 hours) to 31.12.9999 (23:59:59 hours) with a resolution of 100 nanoseconds (this unit is called a ‘tick’). Floating-point (decimal) numbers are stored in binary-coded decimal format (see Section 1.02).

There are many more data types. Programmers can also design and declare their own data types (see Chapter 16 (Section 16.01) and Chapter 26 (Section 26.01).

TASK 14.02

1. Look at the identifier tables in Chapter 12 (Tables 12.02 and 12.04 to 12.12). Decide which data type from your language is appropriate for each variable listed.
2. Write program code to implement the pseudocode from Worked Example 12.01 in Chapter 12.

14.04 Boolean expressions

In Chapter 12 (Section 12.06), we covered logic statements. These were statements that included a condition. Conditions are also known as Boolean expressions and evaluate to either True or False. True and False are known as Boolean values.

Simple Boolean expressions involve comparison operators (see Table 14.04). Complex Boolean expressions also involve Boolean operators (see Table 14.05).

Operation	**Pseudocode**	**Python**	**VB.NET**	**Java**
equal	`=`	`==`	`=`	`==`
not equal	`<>`	`!=`	`<>`	`!=`
greater than	`>`	`>`	`>`	`>`
less than	`<`	`<`	`<`	`<`
greater than or equal to	`>=`	`>=`	`>=`	`>=`
less than or equal to	`<=`	`<=`	`<=`	`<=`

Table 14.04 Comparison operators

Operation	**Pseudocode**	**Python**	**VB.NET**	**Java**
AND (logical conjunction)	`AND`	`and`	`And`	`&&`
OR (logical inclusion)	`OR`	`or`	`Or`	`||`
NOT (logical negation)	`NOT`	`not`	`Not`	`!`

Table 14.05 Boolean operators

14.05 Selection

IF…THEN statements

In pseudocode the IF…THEN construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
ENDIF
```

Syntax definitions

Python

```
if <Boolean expression>:
  <statement(s)>
```

Note that the `THEN` keyword is replaced by a colon (:). Indentation is used to show which statements form part of the conditional statement.

VB.NET

```
If <Boolean expression> Then
  <statement(s)>
End If
```

Note the position of `Then` on the same line as the Boolean expression. The `End If` keywords should line up with the `If` keyword.

Java

```
if (<Boolean expression>)
  <statement>;
```

Note that the Boolean expression is enclosed in brackets.
If more than one statement is required as part of the conditional statement, the statements must be enclosed in braces { }.

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
ENDIF
```

Code examples

Python

```
if x < 0:
  print("Negative")
```

VB.NET

```
If x < 0 Then
  Console.WriteLine("Negative")
End If
```

Java

```
if (x < 0)
  System.out.println("Negative");
```

TASK 14.03

Write program code to implement the pseudocode from Worked Example 12.03 in Chapter 12.

IF…THEN…ELSE statements

In pseudocode, the IF…THEN…ELSE construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    <statement(s)>
ENDIF
```

Syntax definitions

Python

```
if <Boolean expression>:
  <statement(s)>
else:
  <statement(s)>
```

Indentation is used to show which statements form part of the conditional statement; the `else` keyword must line up with the corresponding `if` keyword.

VB.NET

```
If <Boolean expression> Then
  <statement(s)>
Else
  <statement(s)>
End If
```

The `Else` keyword is on its own on a separate line. It is good programming practice to line it up with the corresponding `If` keyword and indent the statements within the conditional statement.

Java

```
if (<Boolean expression>)
  <statement>;
else
  <statement>;
```

If more than one statement is required in the `else` part of the statement, the statements must be enclosed in braces `{ }`.

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    OUTPUT "Positive"
ENDIF
```

Code examples

Python

```
if x < 0:
  print("Negative")
else:
  print("Positive")
```

VB.NET

```
If x < 0 Then
  Console.WriteLine("Negative")
Else
  Console.WriteLine("Positive")
End If
```

Java

```
if (x < 0)
  System.out.println("Negative");
else
  System.out.println("Positive");
```

Nested IF statements

In pseudocode, the nested IF statement is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    IF <Boolean expression>
      THEN
        <statement(s)>
      ELSE
        <statement(s)>
    ENDIF
ENDIF
```

Syntax definitions

Python

```
if <Boolean expression>:
  <statement(s)>
elif <Boolean expression>:
```

Note the keyword `elif` (an abbreviation of `else if`). This keyword must line up

```
  <statement(s)>
else:
  <statement(s)>
```

with the corresponding `if`.
There can be as many `elif` parts to this construct as required.

VB.NET

```
If <Boolean expression> Then
  <statement(s)>
ElseIf
  <statement(s)>
Else
  <statement(s)>
End If
```

If `ElseIf` is used as one word, only one `End If` is required at the end of this construct.
There can be as many `ElseIf` parts as required.

Java

```
if (<Boolean expression>)
  <statement>;
else if (<Boolean expression>)
  <statement>;
else
  <statement>;
```

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    IF x = 0
      THEN
        OUTPUT "Zero"
      ELSE
        OUTPUT "Positive"
    ENDIF
ENDIF
```

Code examples

Python

```
if x < 0:
  print("Negative")
elif x == 0:
   print("Zero")
else:
   print("Positive")
```

VB.NET

```
If x < 0 Then
   Console.WriteLine("Negative")
ElseIf x = 0 Then
   Console.WriteLine("Zero")
Else
   Console.WriteLine("Positive")
End If
```

Java

```
if (x < 0)
{
   System.out.println('Negative');
}
else if (x == 0)
{
   System.out.println('Zero');
}
else
{
   System.out.println('Positive');
}
```

TASK 14.04

Write program code to implement the pseudocode from Worked Example 12.02 in Chapter 12.

CASE statements

An alternative selection construct is the CASE statement. Each considered CASE condition can be:

- a single value
- single values separated by commas
- a range.

In pseudocode, the CASE statement is written as:

```
CASE OF <expression>
  <value1>             : <statement(s)>
  <value2>,<value3>    : <statement(s)>
  <value4> TO <value5> : <statement(s)>
  .
  .
  OTHERWISE <statement(s)>
ENDCASE
```

The value of `<expression>` determines which statements are executed. There can be as many separate cases as required. The `OTHERWISE` clause is optional and useful for error trapping.

Syntax definitions

Python	Python does not have a CASE statement. You need to use nested If statements instead.
VB.NET	

```
Select Case <expression>
  Case value1
    <statement(s)>
  Case value2,value3
    <statement(s)>
  Case value4 To value5
    <statement(s)>
  .
  .
   .
  Case Else
    <statement(s)>
End Select
```

Java

```
switch (<expression>)
{
  case value1:
    <statement(s)>;
    break;
  case value2: case value3:
    <statement(s)>;
    break;
  .
  .
  .
  default: <statement(s)>;
}
```

In pseudocode, an example CASE statement is:

```
CASE OF Grade
  "A"      : OUTPUT "Top grade"
  "F", "U" : OUTPUT "Fail"
  "B".."E" : OUTPUT "Pass"
OTHERWISE OUTPUT "Invalid grade"
ENDCASE
```

Code examples

Python

```
if Grade == "A":
  print("Top grade")
elif Grade == "F" or Grade == "U":
  print("Fail")
elif Grade in ("B", "C", "D", "E"):
  print("Pass")
else:
  print("Invalid grade")
```

VB.NET

```
Select Case Grade
  Case "A"
    Console.WriteLine("Top grade")
  Case "F","U"
    Console.WriteLine("Fail")
  Case "B" To "E"
    Console.WriteLine("Pass")
  Case Else
    Console.WriteLine("Invalid grade")
End Select
```

Java

```
switch (grade)
{
  case 'A':
    System.out.println("Top Grade");
    break;
  case 'F': case 'U':
    System.out.println("Fail");
    break;
  case 'B': case 'C': case 'D': case 'E':
    System.out.println("Pass");
    break;
  default:
    System.out.println("Invalid grade");
}
```

TASK 14.05

The problem to be solved: the user enters the number of the month and year. The output is the number of days in that month. The program has to check if the year is a leap year for February.

The pseudocode solution is:

```
INPUT MonthNumber
INPUT Year
Days ← 0
CASE OF MonthNumber
  CASE 1,3,5,7,8,10,12 : Days ← 31
  CASE 4,6,9,11 : Days ← 30
  CASE 2 : Days ← 28
    If Year MOD 400 = 0
      THEN // it is a leap year
        Days ← 29
    ENDIF
    IF (Year MOD 4 = 0) AND (Year MOD 100 > 0)
      THEN // it is a leap year
        Days ← 29
    ENDIF
  OTHERWISE OUTPUT "Invalid month number"
ENDCASE
OUTPUT Days
```

Write program code to implement the pseudocode above.

14.06 Iteration

Count-controlled (FOR) loops

In pseudocode, a count-controlled loop is written as:

```
FOR <control variable> ← s TO e STEP i // STEP is optional
    <statement(s)>
NEXT <control variable>
```

The control variable starts with value `s`, increments by value `i` each time round the loop and finishes when the control variable reaches the value `e`.

Syntax definitions

Python

```
for <control variable> in range(s, e, i):
    <statement(s)>
```

The values `s`, `e` and `i` must be of type integer.
The loop finishes when the control variable is just below `e`.
The values for `s` and `i` can be omitted and they default to 0 and 1, respectively.

VB.NET

```
For <control variable> = s To e Step i
    <statement(s)>
Next
```

The values `s`, `e` and `i` can be of type integer or float.

Java

```
for (int i = s; i < e; i++)
    <statement>;
```

Where `i` is the control variable.

In pseudcode, examples are:

```
FOR x ← 1 TO 5
    OUTPUT x
NEXT x
FOR x = 2 TO 14 STEP 3
    OUTPUT x
NEXT x
FOR x = 5 TO 1 STEP -1
    OUTPUT x
NEXT x
```

Code examples

Python

```
for x in range(5):
    print(x, end=' ')
```

The start value of x is 0 and it increases by 1 on each iteration.
Output: `0 1 2 3 4`

```
for x in range(2, 14, 3):
    print(x, end=' ')
```

Output: `2 5 8 11`

```
for x in range(5, 1, -1):
    print(x, end=' ')
```

The start value of x is 5 and it decreases by 1 on each iteration.
Output: `5 4 3 2`

```
for x in ["a", "b", "c"]:
    print(x, end='')
```

The control variable takes the value of each of the group elements in turn.
Output: `abc`

VB.NET

```
For x = 1 To 5
    Console.Write(x)
Next
```

Output: `1 2 3 4 5`

```
For x = 2 To 14 Step 3
```

Output: `2 5 8 11 14`

```
    Console.Write(x)
Next
```

```
For x = 5 To 1 Step -1
    Console.Write(x)
Next
```

Output: `5 4 3 2 1`

```
For x = 1 To 2.5 Step 0.5
    Console.WriteLine(x)
Next
```

```
Output:
1
1.5
2
2.5
```

```
For Each x In {"a", "b", "c"}
    Console.Write(x)
Next
```

The control variable takes the value of each of the group elements in turn.
Output: `abc`

Java

```
for (int x = 1; x < 6; x++)
{
    System.out.print(x);
}
```

Output: `12345`

```
for (int x = 2; x < 15; x = x
+ 3)
{
    System.out.print(x +
"  ");
}
```

Output: `2   5   8   11   14`

```
for (int x = 5; x > 0; x--)
{
    System.out.print(x +
"  ");
}
```

Output: `5   4   3   2   1`

```
for (double x = 1; x < 3; x =
x + 0.5)
{
    System.out.print(x +
"  ");
}
```

Output: `1.0   1.5   2.0   2.5`

```
char[] letter = {'a', 'b',
'c'};
for (char x : letter )
{
   System.out.print(x);
}
```

The control variable takes the value of each of the group elements in turn.
Output: `abc`

TASK 14.06

1. Write program code to implement the pseudocode from Worked Example 12.05 in Chapter 12.
2. Write program code to implement the pseudocode from Worked Example 12.08 in Chapter 12.
3. Write program code to implement the pseudocode from Worked Example 12.09 in Chapter 12.

Post-condition loops

A post-condition loop, as the name suggests, executes the statements within the loop at least once. When the condition is encountered, it is evaluated. As long as the condition evaluates to False, the statements within the loop are executed again. When the condition evaluates to True, execution will go to the next statement after the loop.

When coding a post-condition loop, you must ensure that there is a statement within the loop that will at some point change the end condition to True. Otherwise the loop will execute forever.

In pseudocode, the post-condition loop is written as:

```
REPEAT
    <statement(s)>
UNTIL <condition>
```

Syntax definitions

Python

Post-condition loops are not available in Python. Use a pre-condition loop instead.

VB.NET

```
Do
    <statement(s)>
Loop Until <condition>
```

Java

```
do
{
    <statement(s)>
} while <condition>;
```

Pseudocode example:

```
REPEAT
    INPUT "Enter Y or N: " Answer
UNTIL Answer = "Y"
```

Code examples

VB.NET

```
Do
   Console.Write("Enter Y or N: ")
   Answer = Console.ReadLine()
Loop Until Answer = "Y"
```

Java

```
do
{
   System.out.print("Enter Y or N: ");
   answer = console.next();
} while  (!(answer.equals("Y")));
```

TASK 14.07

1 Write program code to implement the pseudocode from Worked Example 12.04 in Chapter 12.

2 Write program code to implement the first algorithm from Worked Example 12.06 in Chapter 12.

Pre-condition loops

Pre-condition loops, as the name suggests, evaluate the condition before the statements within the loop are executed. Pre-condition loops will execute the statements within the loop as long as the condition evaluates to True. When the condition evaluates to False, execution will go to the next statement after the loop. Note that any variable used in the condition must not be undefined when the loop structure is first encountered.

When coding a pre-condition loop, you must ensure that there is a statement within the loop that will at some point change the value of the controlling condition. Otherwise the loop will execute forever.

In pseudocode the pre-condition loop is written as:

```
WHILE <condition> DO
    <statement(s)>
ENDWHILE
```

Syntax definitions

Python

```
while <condition>:
    <statement(s)>
```

Note that statements within the loop must be indented by a set number of spaces. The first statement after the loop must be indented less.

VB.NET

```
Do While <condition>
    <statement(s)>
Loop
Do Until <condition>
    <statement(s)>
Loop
```

Note the keyword `Loop` indicates the end of the loop.
VB.NET also has a pre-condition `Until` loop. This will execute the statements within the loop as long as the condition evaluates to False. If the condition evaluates to True when the loop is first encountered, the statements within the loop are not executed at all.

Java

```
while (<condition>)
{
    <statement(s)>;
}
```

Pseudocode example,

```
Answer ← ""
WHILE Answer <> "Y" DO
    INPUT "Enter Y or N: " Answer
ENDWHILE
```

Code examples

Python

```
Answer = ''
while Answer != 'Y':
   Answer = input("Enter Y or N: ")
```

VB.NET

```
Dim Answer As String = ""
Do While Answer <> "Y"
   Console.Write("Enter Y or N: ")
   Answer = Console.ReadLine()
Loop
Answer = ""
Do Until Answer = "Y"
   Console.Write("Enter Y or N: ")
   Answer = Console.ReadLine()
Loop
```

Java

```
String answer = "";
while(answer.equals("Y") == false)
{
   System.out.print("Enter Y or N: ");
   answer = console.next();
}
```

TASK 14.08

Write program code to implement the second algorithm from Worked Example 12.06 in Chapter 12.

Which loop structure to use?

If you know how many times around the loop you need to go when the program execution gets to the loop statements, use a count-controlled loop. If the termination of the loop depends on some condition determined by what happens within the loop, then use a conditional loop. A pre-condition loop has the added benefit that the loop may not be entered at all, if the condition does not require it.

TIP

Computer Scientists like efficient code. Choosing the most suitable types of selection statements and loop structures is an important step along the way to design efficient code.

14.07 Built-in functions

Programming environments provide many built-in functions. Some of them are always available to use; some need to be imported from specialist module libraries.

Discussion Point:

Investigate your own programming environment and research other library routines.

String manipulation functions

Table 14.06 contains some useful functions for manipulating strings.

Description	Pseudocode	Python	VB.NET	Java
Access a single character using its position P in a string `ThisString`	`ThisString[P]` Counts from 1	`ThisString[P]` Counts from 0	`ThisString(P)` Counts from 0	`ThisString.charAt(P)` Counts from 0
Returns the character whose ASCII value is i	`CHR(i : INTEGER) RETURNS CHAR`	`chr(i)`	`Chr(i)`	`(char) i;`
Returns the ASCII value of character ch	`ASC(ch) RETURNS INTEGER`	`ord(ch)`	`Asc(ch)`	`(int) ch;`
Returns the integer value representing the length of string S	`LENGTH(S : STRING) RETURNS INTEGER`	`len(S)`	`len(S)`	`S.length();`
Returns leftmost L characters from S	`LEFT(S : STRING, L : INTEGER) RETURNS STRING`	`S[0:L]` See the next section, on slicing	`Left(S, L)`	`S.subString (0, L)`
Returns rightmost L characters from S	`RIGHT(S: STRING, L : INTEGER) RETURNS STRING`	`S[-L:]` See the next section, on slicing	`Right(S, L)`	`S.subString (S.length() – L)`
Returns a string of length L starting at position P from S	`MID(S : STRING, P : INTEGER, L : INTEGER) RETURNS STRING`	`S[P : P + L]` See the next section, on slicing	`mid(S, P, L)`	`S.subString(P, P + L)`
Returns the character value representing the lower case equivalent of Ch	`LCASE(Ch : CHAR) RETURNS CHAR`	`Ch.lower()`	`LCase(Ch)`	`Character.toLowerCase(ch)`
Returns the character value representing the upper case equivalent of Ch	`UCASE(Ch : CHAR) RETURNS CHAR`	`Ch.upper()`	`UCase(Ch)`	`Character.toUpperCase(ch)`
Returns a string formed by converting all alphabetic characters of S to upper case	`TO_UPPER(S : STRING) RETURNS STRING`	`S.upper()`	`S.ToUpper`	`S.toUpperCase()`
Returns a string formed by converting all alphabetic characters of S to lower	`TO_LOWER(S : STRING) RETURNS STRING`	`S.lower()`	`S.ToLower`	`S.toLowerCase()`

case				
Concatenate (join) two strings	`S1 & S2`	`s = S1 + S2`	`s = S1 + S2` `s = S1 & S2`	`s = S1 + S2;`

Table 14.06 Some useful string manipulation functions

Slicing in Python

In Python a subsequence of any sequence type (e.g. lists and strings) can be created using 'slicing'.

For example, to get a substring of length L from position P in string S we write S[P : P + L].

Figure 13.05 shows a representation of `ThisString`. If we want to return a slice of length 3 starting at position 3, we use `ThisString[3 : 6]` to give 'DEF'. Position is counted from 0 and the position at the upper bound of the slice is not included in the substring.

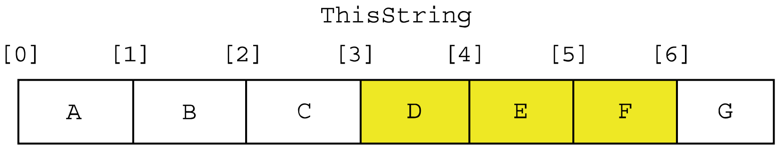


Figure 14.05 A representation of `ThisString`

If you imagine the numbering of each element to start at the left-hand end (as shown in Figure 14.05), then it is easier to see how the left element (the lower bound) is included, but the right element (the upper bound) is excluded. Table 14.07 shows some other useful slices in Python.

Expression	Result	Explanation
`ThisString[2:]`	`CDEFG`	If you do not state the upper bound, the slice includes all characters to the end of the string.
`ThisString[:2]`	`AB`	If you do not state the lower bound, the slice includes all characters from the beginning of the string.
`ThisString[-2:]`	`FG`	A negative lower bound means that it takes the slice starting from the end of the string.
`ThisString[:-2]`	`ABCDE`	A negative upper bound means that it terminates the string at that position.

Table 14.07 Some useful slices in Python

Truncating numbers

Instead of rounding, sometimes we just want the whole number part of a real number.

This is known as 'truncation'.

Pseudocode	`INT(x : REAL) RETURNS INTEGER`	Returns the integer part of `x`.
Python	`int(x)`	If `x` is a floating-point number, the conversion truncates towards zero.
VB.NET	`Math.Truncate(x)`	The whole number part of the real number `x` is returned.
Java	`(int) x;`	Casts the number as an integer.

Converting a string to a number

Sometimes a whole number may be held as a string. To use such a number in a calculation, we first need to convert it to an integer. For example, these functions return the integer value 5 from the string `"5"`:

Python	`int(S)`
VB.NET	`CInt(S)`
Java	`Integer.valueOf(S)`

Sometimes a number with a decimal point may be held as a string. To use such a number in a calculation, we first need to convert it to a real (float). For example, these functions return the real number 75.43 from the string `"75.43"`:

Pseudocode	`STRING_TO_NUM(x : STRING) RETURNS REAL`	Returns a numeric representation of a string.
Python	`float(x)`	The returned value is a floating-point number.
VB.NET	`CDbl(x)`	The returned value is of type double.
Java	`Float.valueOf(x)`	The returned value is a floating-point number.

Random number generator

Random numbers are often required for simulations. Most programming languages have various random number generators available. As the random numbers are generated through a program, they are referred to as 'pseudo-random' numbers. A selection of the most useful random number generators are shown in the following code examples.

Code examples

Python — This code produces a random number between 1 and 6 inclusive.

```
# in the random library:
randint(1, 6)
```

VB.NET — You have to set up a RandomNumber object (see Chapter 27). This code generates an integer between 1 (inclusive) and 6 (exclusive).

```
Dim RandomNumber As New Random
Dim x As Integer
x = RandomNumber.Next(1, 6)
```

Java — You have to set up a RandomNumber object (see Chapter 27). This code generates an integer between 1 (inclusive) and 6 (inclusive).

```
import java.util.Random;
Random randomNumber = new Random();
int x = randomNumber.nextInt(6) + 1;
```

TASK 14.09

1. Write program code to generate 20 random numbers in the range 1 to 10 inclusive.
2. Write program code to implement the pseudocode using a pre-condition loop from Worked Example 12.07 in Chapter 12.

Discussion Point:

What other useful functions can you find? Which module libraries have you searched?

14.08 Procedures

In Chapter 12 (Section 12.09), we used procedures as a means of giving a group of statements a name. When we want to program a procedure we need to define it before the main program. We call it in the main program when we want the statements in the procedure body to be executed.

In pseudocode, a procedure definition is written as:

```
PROCEDURE <procedureIdentifier> // this is the procedure header
    <statement(s)>   // these statements are the procedure body
ENDPROCEDURE
```

This procedure is called using the pseudocode statement:

```
CALL <procedureIdentifier>
```

Syntax definitions

Python

```
def <identifier>():
    <statement(s)>
```

VB.NET

```
Sub <identifier>()
    <statement(s)>
End Sub
```

Java

```
void <identifier>()
{
    <statement(s)>;
}
```

When programming a procedure, note where the definition is written and how the procedure is called from the main program.

Here is an example pseudocode procedure definition:

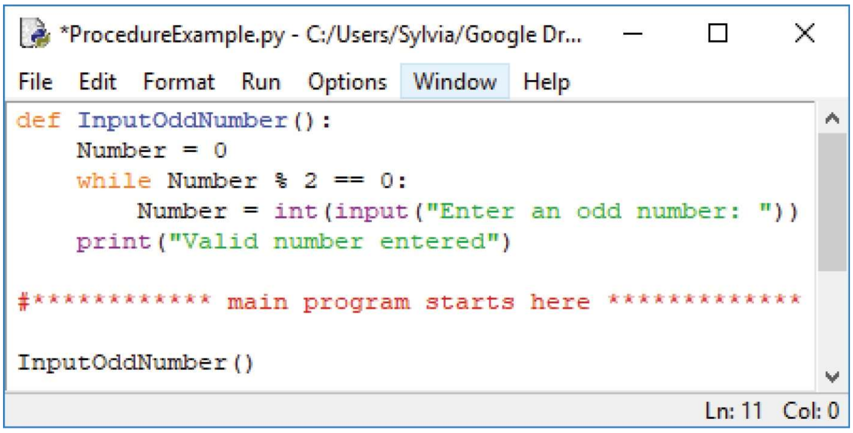
```
PROCEDURE InputOddNumber
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
ENDPROCEDURE
```

This procedure is called using the CALL statement:

```
CALL InputOddNumber
```

Code examples

Python



*ProcedureExample.py - C:/Users/Sylvia/Google Dr...
File Edit Format Run Options Window Help
Ln: 11 Col: 0

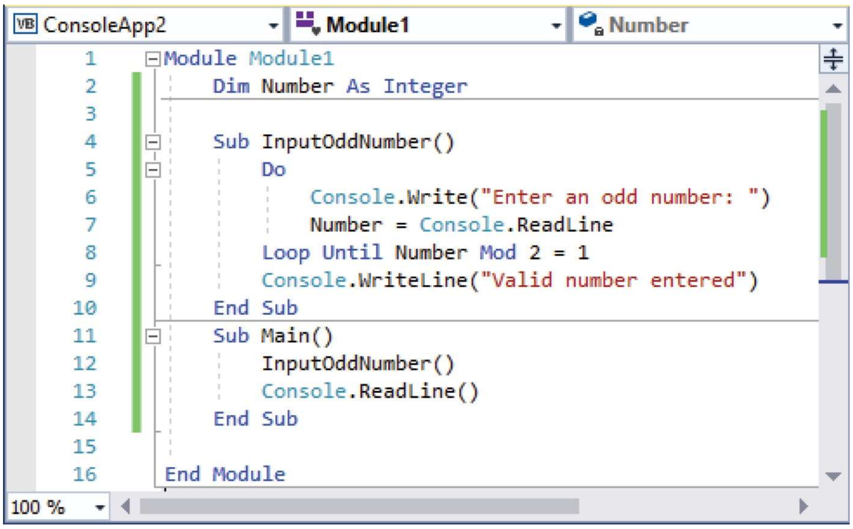
```python
def InputOddNumber():
    Number = 0
    while Number % 2 == 0:
        Number = int(input("Enter an odd number: "))
    print("Valid number entered")

#************ main program starts here *************

InputOddNumber()
```

Figure 14.06 The Python editor with a procedure

The Python editor colour-codes the different parts of a statement. This helps when you are typing your own code. The indentation shows which statements are part of the loop.

The built-in function `input` returns a string, which must be converted to an integer before it can be used as a number.

VB.NET

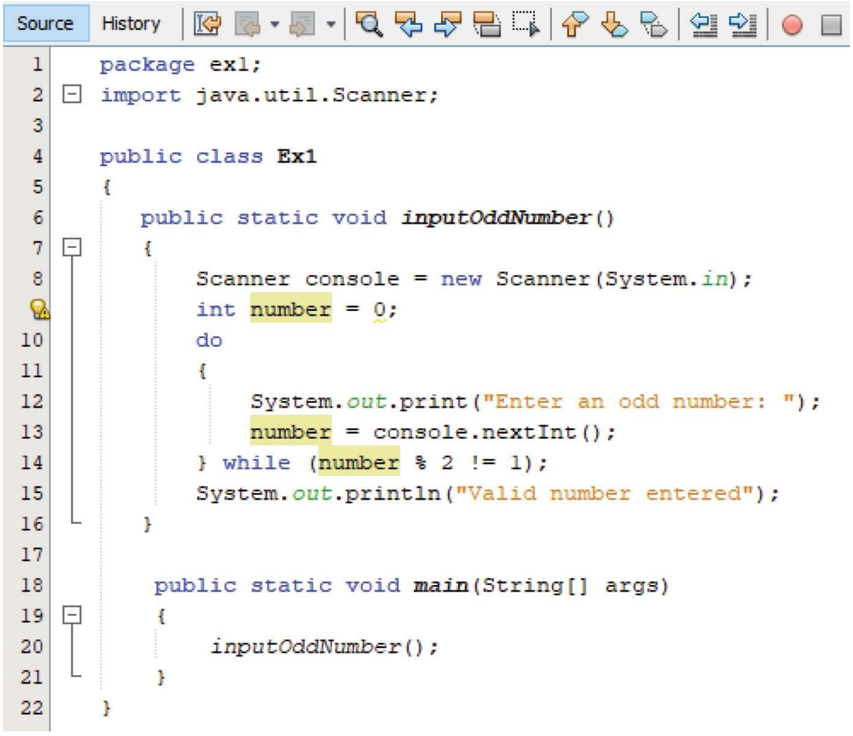


```
Module Module1
    Dim Number As Integer

    Sub InputOddNumber()
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        Console.WriteLine("Valid number entered")
    End Sub
    Sub Main()
        InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.07 The Visual Basic Express editor with a procedure

The Visual Basic Express editor colour-codes different parts of the statement, so it is easy to see if syntax errors are made. The editor also auto-indents and capitalises keywords.
Variables need to be declared before they are used. The editor will follow the capitalisation of the variable declaration when you type an identifier without following your original capitalisation.
The editor is predictive: pop-up lists will show when you type the first part of a statement.
When you execute the Main program, `Console.ReadLine()` keeps the run-time window open.

Java



```
package ex1;
import java.util.Scanner;

public class Ex1
{
    public static void inputOddNumber()
    {
        Scanner console = new Scanner(System.in);
        int number = 0;
        do
        {
            System.out.print("Enter an odd number: ");
            number = console.nextInt();
        } while (number % 2 != 1);
        System.out.println("Valid number entered");
    }

     public static void main(String[] args)
     {
         inputOddNumber();
     }
}
```

Figure 14.08 The NetBeans editor with a void method

The editor automatically colour codes keyword and strings.
The procedure body is enclosed within braces { and }.
The editor is predictive: pop-up lists will show when you type the first part of a statement.

Variables need to be declared before they are used.

TASK 14.10

Write program code to implement the pseudocode from Worked Example 12.11 in Chapter 12.

14.09 Functions

In Section 14.07 we used built-in functions. These are useful subroutines written by other programmers and made available in module libraries. The most-used ones are usually in the system library, so are available without you having to import them.

You can write your own functions. Any function you have written can be used in another program if you build up your own module library.

A function is used as part of an expression. When program execution gets to the statement that includes a function call as part of the expression, the function is executed. The **return value** from this function call is then used in the expression.

When writing your own function, ensure you always return a value as part of the statements that make up the function (the function body). You can have more than one RETURN statement if there are different paths through the function body.

In pseudocode, a function definition is written as:

```
FUNCTION <functionIdentifier> RETURNS <dataType> // function header
    <statement(s)> // function body
    RETURN <value>
ENDFUNCTION
```

Syntax definitions

Python

```
def <functionIdentifier>():
    <statement(s)>
    return <value>
```

VB.NET

```
Function <functionIdentifier>() As <dataType>
    <statement(s)>
    <functionIdentifier> = <value> 'Return <value>
End Function
```

Java

```
<data type> <functionIdentifier>()
{
    <statement(s)>;
    return <value>;
}
```

When programming a function, the definition is written in the same place as a procedure. The function is called from within an expression in the main program, or in a procedure.

Different programming languages use different terminology for their subroutines, as listed in Table 14.08.

Pseudocode	**PROCEDURE**	**FUNCTION**
Python	void function	fruitful function
VB	Subroutine	Function
Java	void method	method

Table 14.08 Programming language terminology for subroutines

Void means ‘nothing’. Both Python and Java use this term to show that their procedure-type subroutine does not return a value. Python refers to both types of subroutines as functions. The fruitful function returns one or more values.

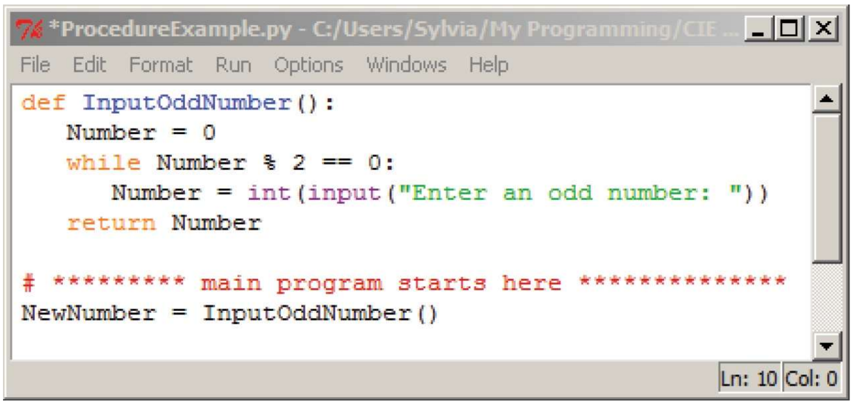
We can write the example procedure from Section 14.09 as a function. In pseudocode, this is:

```
FUNCTION InputOddNumber RETURNS INTEGER
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
    RETURN Number
```

```
ENDFUNCTION
```

Code examples

Python


*ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE ...
File
Edit
Format
Run
Options
Windows
Help
Ln: 10
Col: 0

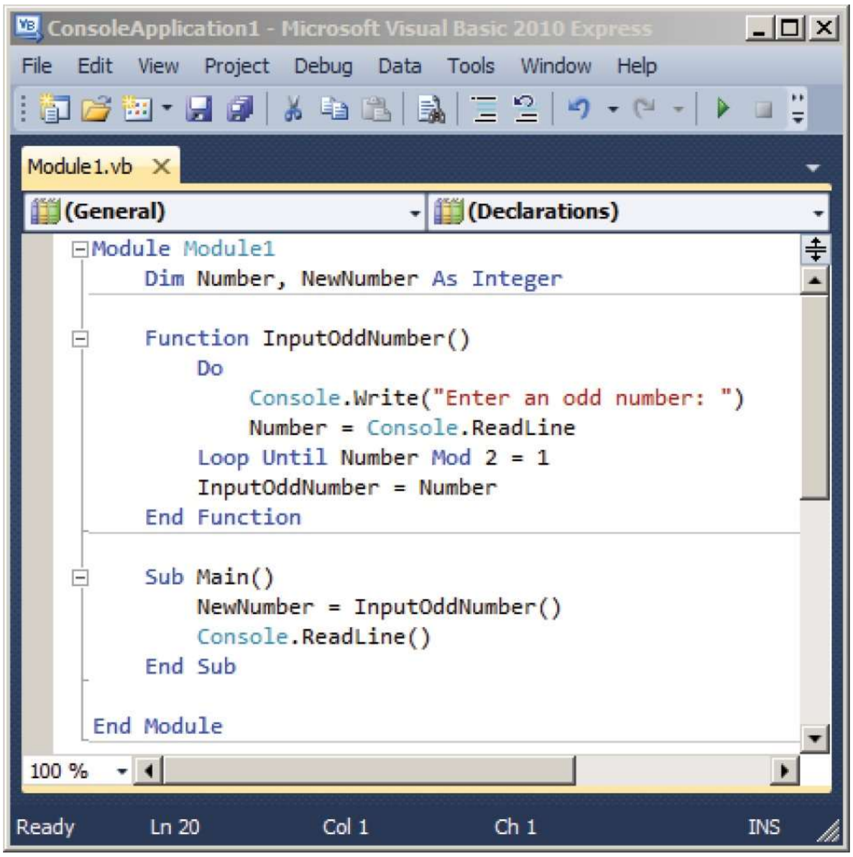
```
def InputOddNumber():
    Number = 0
    while Number % 2 == 0:
        Number = int(input("Enter an odd number: "))
    return Number

# ********* main program starts here **************
NewNumber = InputOddNumber()
```

Figure 14.09 The Python editor with a function and local variable

The variable `Number` in Figure 14.09 is not accessible in the main program. Python’s variables are local unless declared to be global.

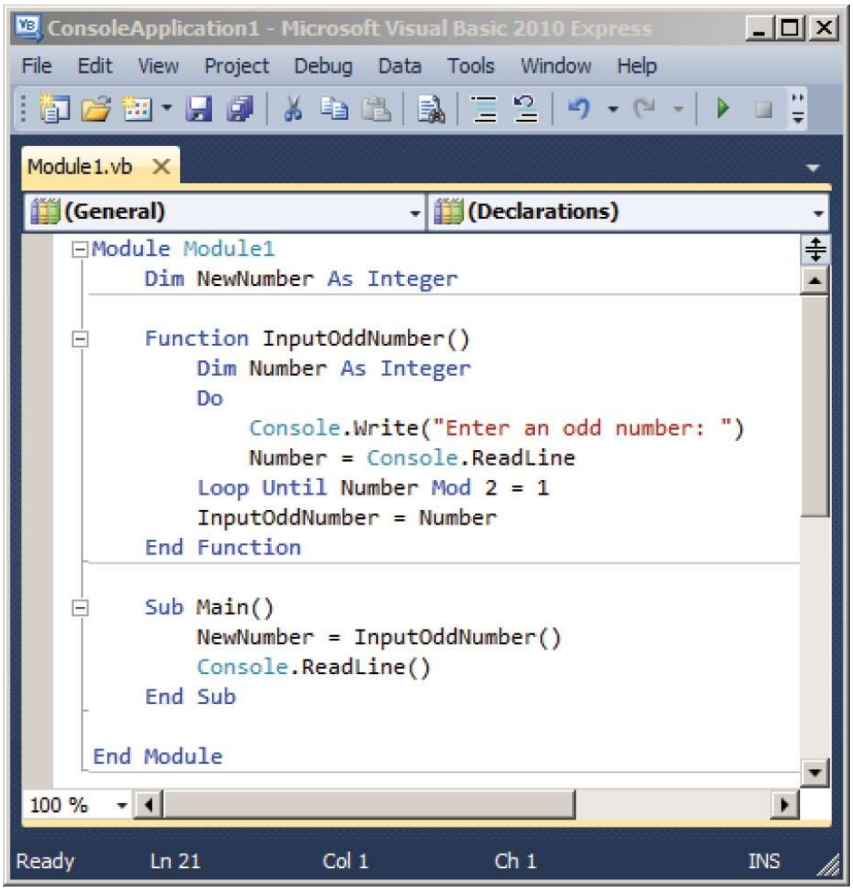
VB.NET

(a)


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File
Edit
View
Project
Debug
Data
Tools
Window
Help
Module1.vb
(General)
(Declarations)
100 %
Ready
Ln 20
Col 1
Ch 1
INS

```
Module Module1
    Dim Number, NewNumber As Integer

    Function InputOddNumber()
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        InputOddNumber = Number
    End Function

    Sub Main()
        NewNumber = InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

(b)

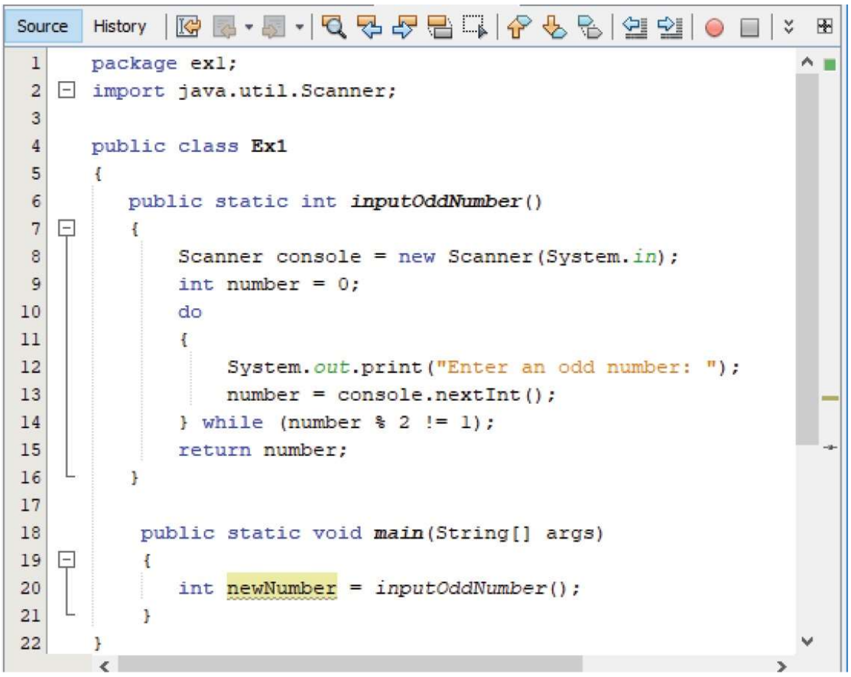

ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %
Ready Ln 21 Col 1 Ch 1 INS

```
Module Module1
    Dim NewNumber As Integer

    Function InputOddNumber()
        Dim Number As Integer
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        InputOddNumber = Number
    End Function

    Sub Main()
        NewNumber = InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.10 The VB.NET editor with (a) global variables and (b) a local variable

The variable `Number` in Figure 14.10(a) is declared as a global variable at the start of the module. This is not good programming practice.	In Figure 14.10(b), the variable `Number` is declared as a local variable within the function.

Java


Source
History

```
package ex1;
import java.util.Scanner;

public class Ex1
{
    public static int inputOddNumber()
    {
        Scanner console = new Scanner(System.in);
        int number = 0;
        do
        {
            System.out.print("Enter an odd number: ");
            number = console.nextInt();
        } while (number % 2 != 1);
        return number;
    }

     public static void main(String[] args)
     {
        int newNumber = inputOddNumber();
     }
}
```

Figure 14.11 The NetBeans editor with a function and local variable

The variable `number` in Figure 14.11 is not accessible in the main program.

A global variable is available in any part of the program code. It is good programming practice to declare a variable that is only used within a subroutine as a local variable.

In Python, every variable is local, unless it is overridden with a global declaration. In VB.NET you need to write the declaration statement for a local variable within the subroutine. Java does not support global variables. However, static variables declared in a class are accessible throughout the class.

TASK 14.11

Write program code to implement the pseudocode from Worked Example 13.05 in Chapter 13. Which variables are global and which are local?

14.10 Passing parameters to subroutines

When a subroutine requires one or more values from the main program, we supply these as **arguments** to the subroutine at call time. This is how we use built-in functions. We don’t need to know the identifiers used within the function when we call a built-in function.

When we define a subroutine that requires values to be passed to the subroutine body, we use a parameter list in the subroutine header. When the subroutine is called, we supply the arguments in brackets. The arguments supplied are assigned to the corresponding **parameter** of the subroutine (note the order of the parameters in the parameter list must be the same as the order in the list of arguments). This is known as the **subroutine interface**.

14.11 Passing parameters to functions

The **function header** is written in pseudocode as:

```
FUNCTION <functionIdentifier> (<parameterList>) RETURNS <dataType>
```

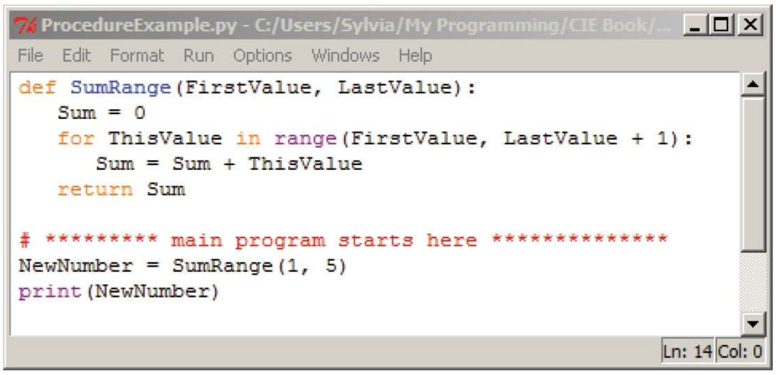
where `<parameterList>` is a list of identifiers and their data types, separated by commas.

Here is an example pseudocode function definition that uses parameters:

```
FUNCTION SumRange(FirstValue : INTEGER, LastValue : INTEGER) RETURNS INTEGER
    DECLARE Sum, ThisValue : INTEGER
    Sum ← 0
    FOR ThisValue ← FirstValue TO LastValue
        Sum ← Sum + ThisValue
    NEXT ThisValue
    RETURN Sum
ENDFUNCTION
```

Code examples

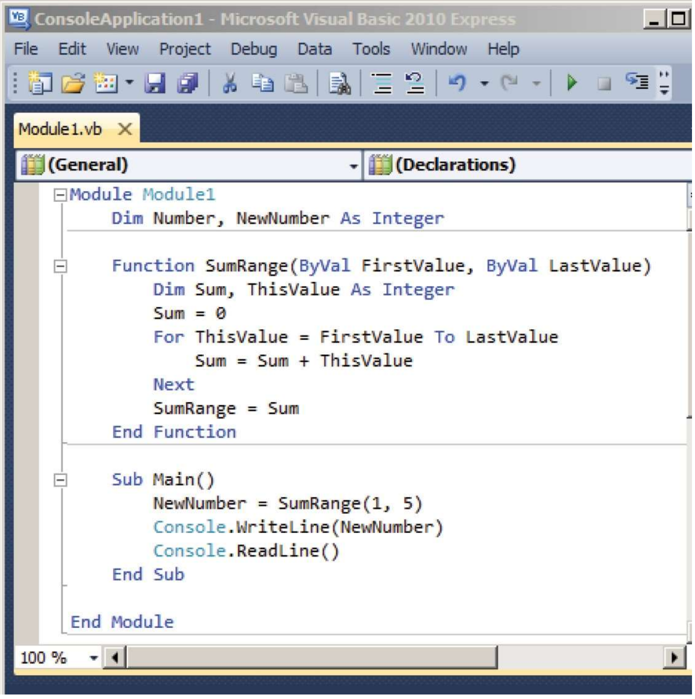
Python


ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE Book/...
File Edit Format Run Options Windows Help
Ln: 14 Col: 0

```
def SumRange(FirstValue, LastValue):
    Sum = 0
    for ThisValue in range(FirstValue, LastValue + 1):
        Sum = Sum + ThisValue
    return Sum

# ********* main program starts here **************
NewNumber = SumRange(1, 5)
print(NewNumber)
```

Figure 14.12 The `SumRange()` function in Python

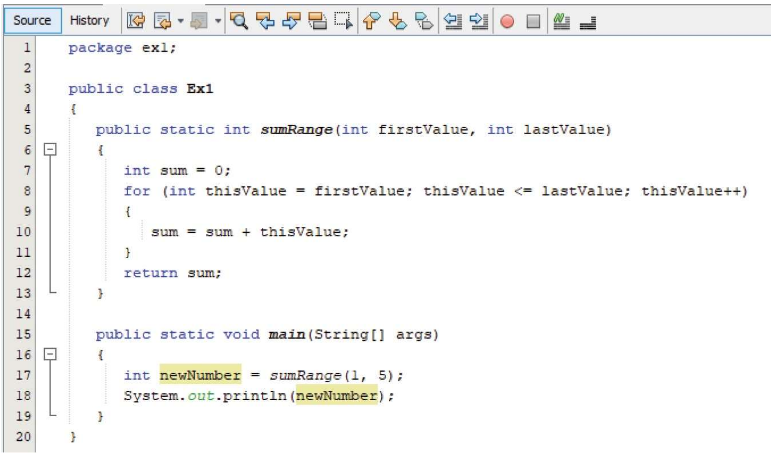
VB.NET


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %

```
Module Module1
    Dim Number, NewNumber As Integer

    Function SumRange(ByVal FirstValue, ByVal LastValue)
        Dim Sum, ThisValue As Integer
        Sum = 0
        For ThisValue = FirstValue To LastValue
            Sum = Sum + ThisValue
        Next
        SumRange = Sum
    End Function

    Sub Main()
        NewNumber = SumRange(1, 5)
        Console.WriteLine(NewNumber)
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.13 The `SumRange()` function in VB.NET

Java



```
package ex1;

public class Ex1
{
    public static int sumRange(int firstValue, int lastValue)
    {
        int sum = 0;
        for (int thisValue = firstValue; thisValue <= lastValue; thisValue++)
        {
            sum = sum + thisValue;
        }
        return sum;
    }

    public static void main(String[] args)
    {
        int newNumber = sumRange(1, 5);
        System.out.println(newNumber);
    }
}
```

Figure 14.14 The `SumRange()` function in Java

TASK 14.12

Write a function to implement the following pseudocode:

```
FUNCTION Factorial (Number : INTEGER) RETURNS INTEGER
    DECLARE Product : INTEGER
    Product ← 1
    FOR n ← 2 TO Number
        Product ← Product * n
    NEXT n
    RETURN Product
ENDFUNCTION
```

14.12 Passing parameters to procedures

If a parameter is passed **by value**, at call time the argument can be an actual value (as we showed in the code examples in Section 14.11). If the argument is a variable, then a copy of the current value of the variable is passed into the subroutine. The value of the variable in the calling program is not affected by what happens in the subroutine.

For procedures, a parameter can be passed **by reference**. At call time, the argument must be a variable. A pointer to the memory location (the address) of that variable is passed into the procedure. Any changes that are applied to the variable's contents will be effective outside the procedure in the calling program/module.

Note that neither of these methods of parameter passing applies to Python. In Python or Java, the method is called pass by object reference. This is basically an object-oriented way of passing parameters and is beyond the scope of this chapter (objects are dealt with in Chapter 27). The important point is to understand how to program in Python and Java to get the desired effect.

The full **procedure header** is written in pseudocode, in a very similar fashion to that for function headers, as:

```
PROCEDURE <ProcedureIdentifier> (<parameterList>)
```

The parameter list needs more information for a procedure definition. In pseudocode, a parameter in the list is represented in one of the following formats:

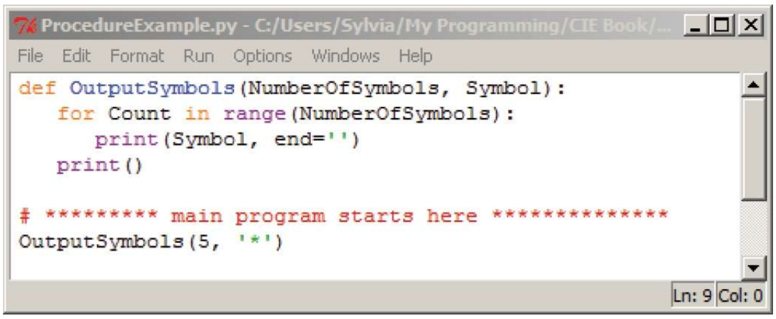
```
BYREF <identifier1> : <dataType>
BYVALUE <identifier2> : <dataType>
```

Passing parameters by value

The pseudocode for the pyramid example in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by value. We can now make that explicit:

```
PROCEDURE OutputSymbols(BYVALUE NumberOfSymbols : INTEGER, Symbol : CHAR)
    DECLARE Count : INTEGER
    FOR Count ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT NewLine
ENDPROCEDURE
```

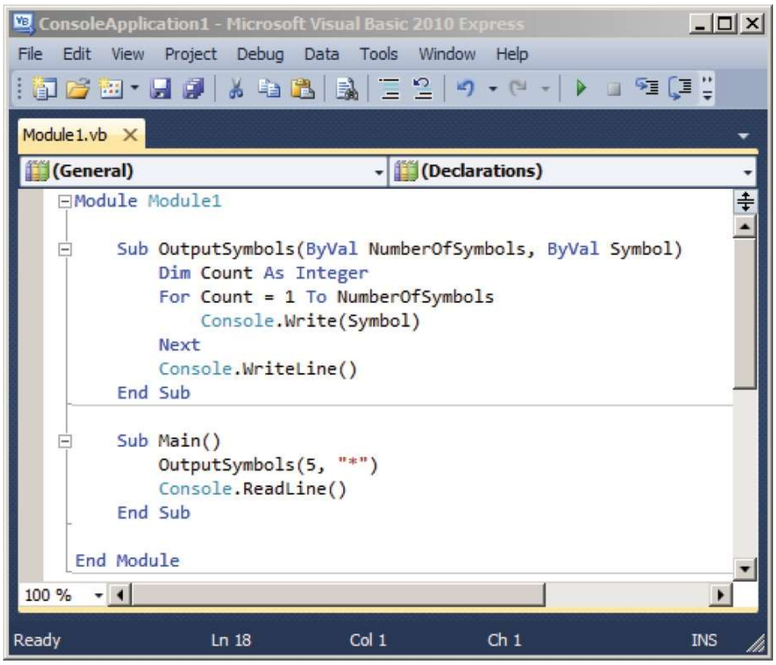
In Python (see Figure 14.15), all parameters behave like local variables and their effect is as though they are passed by value.


ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE Book/...
File
Edit
Format
Run
Options
Windows
Help
Ln: 9
Col: 0

```
def OutputSymbols(NumberOfSymbols, Symbol):
    for Count in range(NumberOfSymbols):
        print(Symbol, end='')
    print()

# ********* main program starts here ***************
OutputSymbols(5, '*')
```

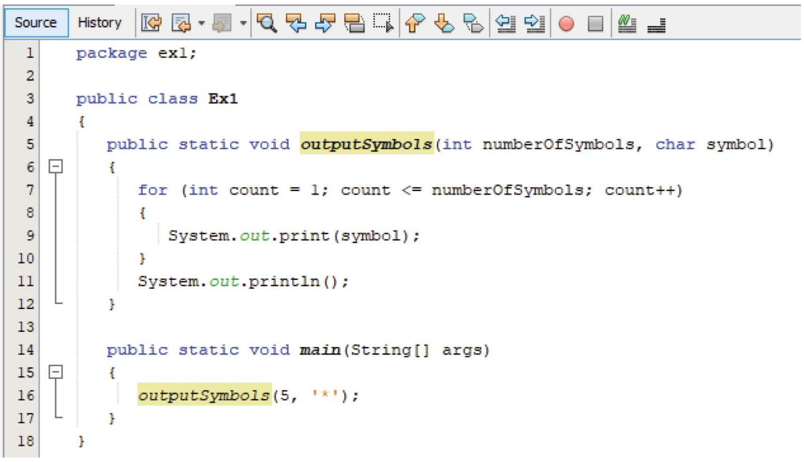
Figure 14.15 Parameters passed to a Python subroutine

In VB.NET (see Figure 14.16), parameters default to passing by value. The keyword ByVal is automatically inserted by the editor.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %
Ready
Ln 18
Col 1
Ch 1
INS

```
Module Module1

    Sub OutputSymbols(ByVal NumberOfSymbols, ByVal Symbol)
        Dim Count As Integer
        For Count = 1 To NumberOfSymbols
            Console.Write(Symbol)
        Next
        Console.WriteLine()
    End Sub

    Sub Main()
        OutputSymbols(5, "*")
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.16 Parameters passed by value to a VB.NET procedure



```
package ex1;

public class Ex1
{
    public static void outputSymbols(int numberOfSymbols, char symbol)
    {
        for (int count = 1; count <= numberOfSymbols; count++)
        {
            System.out.print(symbol);
        }
        System.out.println();
    }

    public static void main(String[] args)
    {
        outputSymbols(5, '*');
    }
}
```

Figure 14.17 Parameters passed by value to a Java procedure

In Java (see Figure 14.17), all parameters behave like local variables and their effect is as though they are passed by value.

Passing parameters by reference

When parameters are passed by reference, when the values inside the subroutine change, this affects the values of the variables in the calling program.

Consider the pseudocode procedure AdjustValuesForNextRow below.

The pseudocode for the pyramid example generated in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by reference. We can now make that explicit:

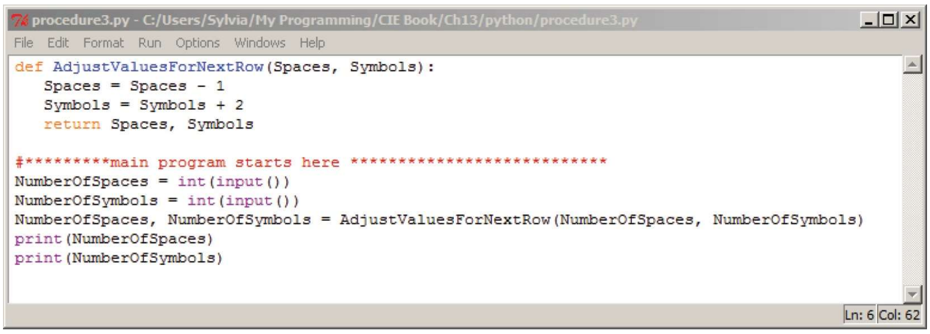
```
PROCEDURE AdjustValuesForNextRow(BYREF Spaces : INTEGER, Symbols : INTEGER)
    Spaces ← Spaces - 1
    Symbols ← Symbols + 2
ENDPROCEDURE
```

The pseudocode statement to call the procedure is:

```
CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
```

The values of the parameters `Spaces` and `Symbols` are changed within the procedure when this is called. The variables NumberOfSpaces and `NumberOfSymbols` in the program code after the call will store the updated values that were passed back from the procedure.

Python does not have a facility to pass parameters by reference. Instead the subroutine behaves as a function and returns multiple values (see Figure 14.18). Note the order of the variables as they receive these values in the main part of the program.

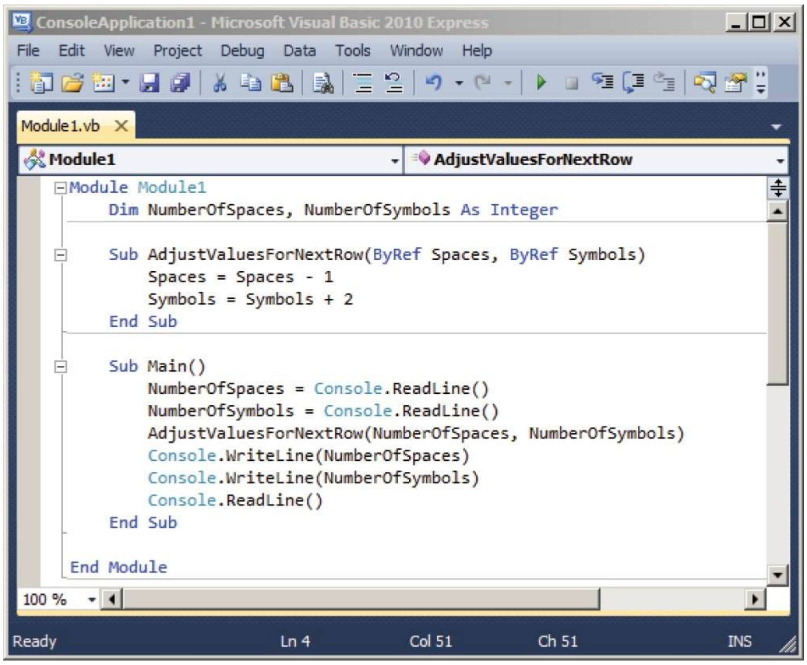

procedure3.py - C:/Users/Sylvia/My Programming/CIE Book/Ch13/python/procedure3.py
File Edit Format Run Options Windows Help
Ln: 6 Col: 62

```
def AdjustValuesForNextRow(Spaces, Symbols):
    Spaces = Spaces - 1
    Symbols = Symbols + 2
    return Spaces, Symbols

#*********main program starts here ***************************
NumberOfSpaces = int(input())
NumberOfSymbols = int(input())
NumberOfSpaces, NumberOfSymbols = AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
print(NumberOfSpaces)
print(NumberOfSymbols)
```

Figure 14.18 Multiple values returned from a Python subroutine

This way of treating a multiple of values as a unit is called a ‘tuple’ (see Chapter 11, Section 11.02). You can find out more by reading the Python help files.

In VB.NET (see Figure 14.19), the ByRef keyword is placed in front of each parameter to be passed by reference.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
AdjustValuesForNextRow
100 %
Ready
Ln 4
Col 51
Ch 51
INS

```
Module Module1
    Dim NumberOfSpaces, NumberOfSymbols As Integer

    Sub AdjustValuesForNextRow(ByRef Spaces, ByRef Symbols)
        Spaces = Spaces - 1
        Symbols = Symbols + 2
    End Sub

    Sub Main()
        NumberOfSpaces = Console.ReadLine()
        NumberOfSymbols = Console.ReadLine()
        AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
        Console.WriteLine(NumberOfSpaces)
        Console.WriteLine(NumberOfSymbols)
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.19 Parameters passed by reference to a VB.NET procedure

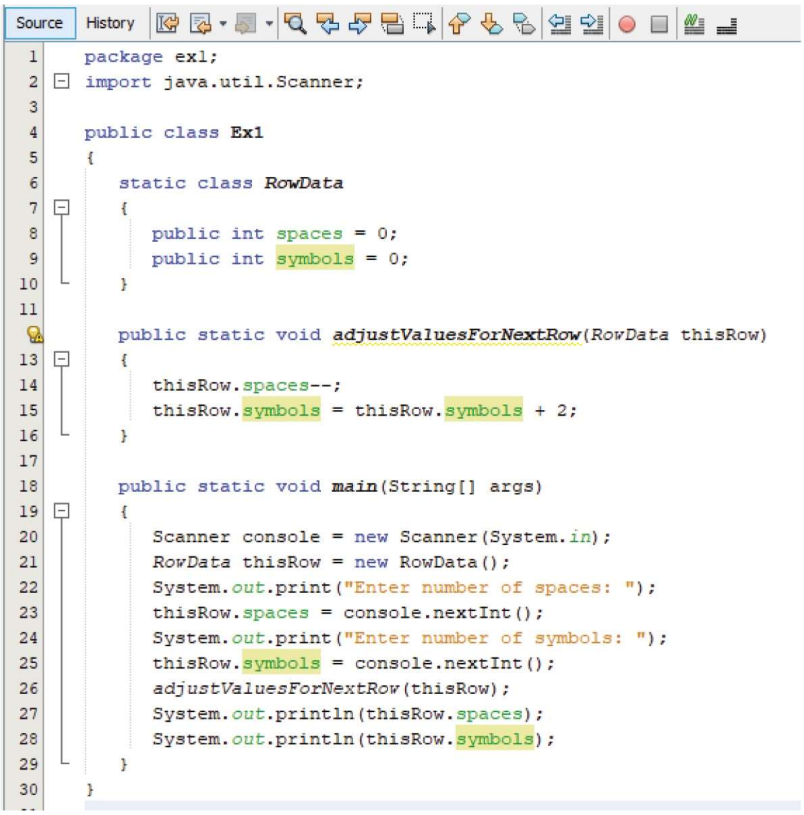
Java does not have a facility to pass simple variable parameters by reference. Only objects can be passed by reference. Arrays are objects in Java, so these are passed by reference.

TIP

If only one value needs to be returned, the subroutine can be written as a function. If more than one value needs to be returned, a work-around is to declare a class and return it as an object (Figure 14.20). If the values to be returned are of the same type, they can be grouped into an array and the array is passed as a reference parameter.

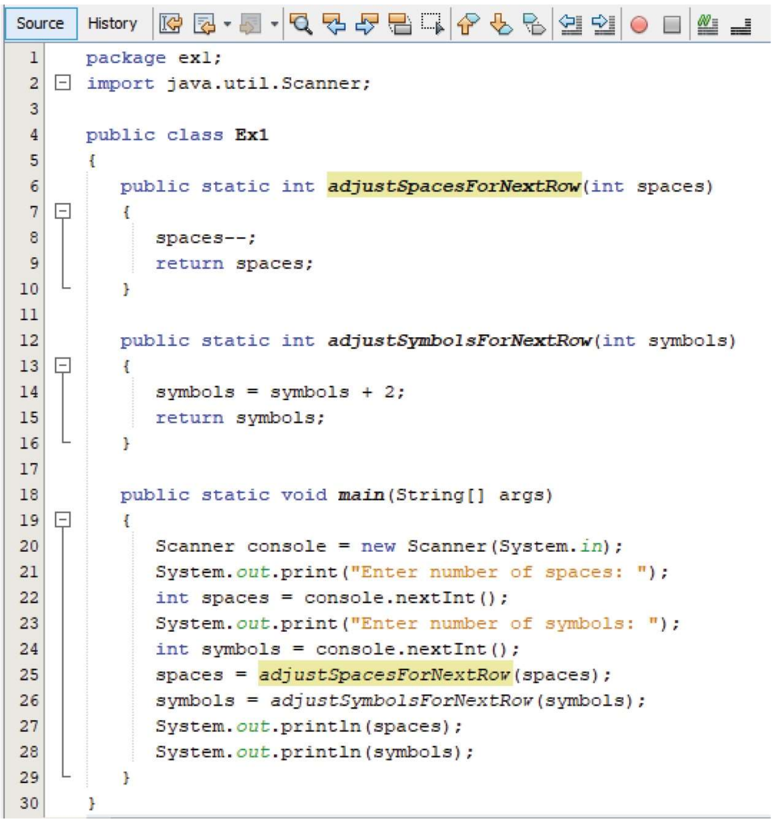
TIP

A preferable solution is to amend the algorithm and write several functions (Figure 14.21).



```java
package ex1;
import java.util.Scanner;

public class Ex1
{
   static class RowData
   {
      public int spaces = 0;
      public int symbols = 0;
   }

   public static void adjustValuesForNextRow(RowData thisRow)
   {
      thisRow.spaces--;
      thisRow.symbols = thisRow.symbols + 2;
   }

   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      RowData thisRow = new RowData();
      System.out.print("Enter number of spaces: ");
      thisRow.spaces = console.nextInt();
      System.out.print("Enter number of symbols: ");
      thisRow.symbols = console.nextInt();
      adjustValuesForNextRow(thisRow);
      System.out.println(thisRow.spaces);
      System.out.println(thisRow.symbols);
   }
}
```

Figure 14.20 Multiple values returned as an object from a Java subroutine



```java
package ex1;
import java.util.Scanner;

public class Ex1
{
   public static int adjustSpacesForNextRow(int spaces)
   {
      spaces--;
      return spaces;
   }

   public static int adjustSymbolsForNextRow(int symbols)
   {
      symbols = symbols + 2;
      return symbols;
   }

   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      System.out.print("Enter number of spaces: ");
      int spaces = console.nextInt();
      System.out.print("Enter number of symbols: ");
      int symbols = console.nextInt();
      spaces = adjustSpacesForNextRow(spaces);
      symbols = adjustSymbolsForNextRow(symbols);
      System.out.println(spaces);
      System.out.println(symbols);
   }
}
```

Figure 14.21 The algorithm changed into two subroutines for Java

14.13 Putting it all together

The programs in this section are full solutions to the pyramid-drawing program developed in Section 14.12.

The parameters of the subroutines have different identifiers from the variables in the main program. This is done deliberately, so that it is quite clear that the parameters and local variables within a subroutine are separate from those in the calling program or module. If a parameter is passed by reference to a procedure, the parameter identifier within the procedure references the same memory location as the variable identifier passed to the procedure as argument.

The pyramid-drawing program in Python, VB.NET and Java

Python

```
SPACE = ' ' # constant to give a space a name
def InputMaxNumberOfSymbols():
   Number = 0
   while Number % 2 == 0:
      print("How many symbols make the base? ")
      Number = int(input("Input an odd number: "))
   return Number

def SetValues():
   Symbol = input("What symbol do you want to use? ")
   MaxSymbols = InputMaxNumberOfSymbols()
   Spaces = (MaxSymbols + 1) // 2
   Symbols = 1
   return Symbol, MaxSymbols, Spaces, Symbols

def OutputChars(Number, Symbol):
   for Count in range (Number):
      print(Symbol, end='')

def AdjustValuesForNextRow(Spaces, Symbols):
   Spaces = Spaces – 1
   Symbols = Symbols + 2
   return Spaces, Symbols

def main():
   ThisSymbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols = SetValues()
   while NumberOfSymbols <= MaxNumberOfSymbols:
      OutputChars(NumberOfSpaces, SPACE)
      OutputChars(NumberOfSymbols, ThisSymbol)
      print()  # move to new line
      NumberOfSpaces, NumberOfSymbols = AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
main()
```

VB.NET

```
Module Module1
   Const Space = " " 'constant to give a space a name
   Dim NumberOfSpaces, NumberOfSymbols As Integer
   Dim MaxNumberOfSymbols As Integer
   Dim ThisSymbol As Char

Sub InputMaxNumberOfSymbols(ByRef Number As Integer)
   Do
      Console.WriteLine("How many symbols make the base? ")
      Console.Write("Input an odd number: ")
      Number = Console.ReadLine()
   Loop Until (Number Mod 2 = 1)
End Sub

Sub SetValues(ByRef Symbol, ByRef MaxSymbols, ByRef Spaces, ByRef Symbols)
   Console.Write("What symbol do you want to use? ")
   Symbol = Console.ReadLine()
   InputMaxNumberOfSymbols(MaxSymbols)
   Spaces = (MaxSymbols + 1) \ 2
   Symbols = 1
End Sub
```

```
Sub OutputChars(ByVal Number, ByVal Symbol)
   Dim Count As Integer
   For Count = 1 To Number
      Console.Write(Symbol)
   Next
End Sub

Sub AdjustValuesForNextRow(ByRef Spaces, ByRef Symbols)
   Spaces = Spaces - 1
   Symbols = Symbols + 2
End Sub

Sub Main()
   SetValues(ThisSymbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
   Do
      OutputChars(NumberOfSpaces, Space)
      OutputChars(NumberOfSymbols, ThisSymbol)
      Console.WriteLine()'move to new line
      AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
   Loop Until NumberOfSymbols > MaxNumberOfSymbols
   Console.ReadLine()
End Sub

End Module
```

Java

```
package ex1;
import java.util.Scanner;

public class Ex1
{
   static final char SPACE = ' ';

   public static char getSymbol()
   {
      Scanner console = new Scanner(System.in);
      System.out.print("What symbol do you want to use? ");
      String response = console.next();
      return response.charAt(0);
   }
   public static int inputMaxNumberOfSymbols()
   {
      Scanner console = new Scanner(System.in);
      int number = 0;
      while ((number % 2) == 0)
      {
         System.out.print("How many symbols make the base? ");
         number = console.nextInt();
      }
      return number;
   }

   public static void outputChars(int number, char symbol)
   {
      for (int count = 0; count < number; count++)
      {
         System.out.print(symbol);
      }
   }
   public static int adjustSpacesForNextRow(int spaces)
   {
      spaces--;
      return spaces;
   }
   public static int adjustSymbolsForNextRow(int symbols)
   {
      symbols = symbols + 2;
      return symbols;
   }
   public static void main(String[] args)
   {
      char thisSymbol = getSymbol();
```

```
        int maxNumberOfSymbols = inputMaxNumberOfSymbols();
        int numberOfSpaces = (maxNumberOfSymbols + 1)/ 2;
        int numberOfSymbols = 1;

        while (numberOfSymbols <= maxNumberOfSymbols)
        {
           outputChars(numberOfSpaces, SPACE);
           outputChars(numberOfSymbols, thisSymbol);
           System.out.println();
           numberOfSpaces = adjustSpacesForNextRow(numberOfSpaces);
           numberOfSymbols = adjustSymbolsForNextRow(numberOfSymbols);
        }
     }
  }
```

Discussion Point:

Can you see how the two procedures OutputSpaces and OutputSymbols have been replaced by a single procedure OutputChars without changing the effect of the program?

14.14 Arrays

Creating 1D arrays

VB.NET, Python and Java number array elements from 0 (the lower bound).

In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Syntax definitions

Python	In Python, there are no arrays. The equivalent data structure is called a list. A list is an ordered sequence of items that do not have to be of the same data type.
VB.NET	`Dim <arrayIdentifier>(<upperBound>) As <dataType>`
Java	`<datatype>[] <arrayIdentifier>;` `<arrayIdentifier> = new int[<upperbound>+1];`

Pseudocode example:

```
  DECLARE List1 : ARRAY[1:3]   OF STRING  // 3 elements in this list
  DECLARE List2 : ARRAY[0:5]   OF INTEGER // 6 elements in this list
  DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
  DECLARE List4 : ARRAY[0:25] OF STRING  // 26 elements in this list
```

Code examples

Python	`List1 = []`
	`List1.append("Fred")`
	`List1.append("Jack")`
	`List1.append("Ali")`
		As there are no declarations, the only way to generate a list is to initialise one. You can append elements to an existing list.

	`List2 = [0, 0, 0, 0, 0, 0]`	You can enclose the elements in `[ ]`.
	`List3 = [0 for i in range(100)]`	You can use a loop.
	`AList = [""] * 26`	You can provide an initial value, multiplied by number of elements required.

VB.NET	`Dim List1 As String () = {"","",""}`
	`Dim List2(5) As Integer`
	`Dim List3(100) As Integer`
	`Dim AList(0 To 25) As String`
		You can initialise an array at declaration time (as with `List1`). Note that `List3` has 101 elements. You can use a range as an array dimension (as with `AList`) however the lower bound must be 0.

Java	`String[] list1 = {"","",""};`
	`int[] list2;`
	`list2 = new int[5];`
	`int[] list3;`
	`list3 = new int[100];`
	`String[] aList;`
	`aList = new String[25];`
		You can initialise an array at declaration time (as with `list1`).

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
  <arrayIdentifier>[x]
```

Pseudocode example:

```
  NList[25] = 0  // set 25th element to zero
  AList[3] = "D" // set 3rd element to letter D
```

Code examples

Python	`NList[24] = 0` `AList[3] = "D"`
VB.NET	`NList(25) = 0` `AList(3) = "D"`
Java	`nList[25] = 0;` `aList[3] = "D";`

In Python, you can print the whole contents of a list using `print(List)`. In VB.NET and Java, you need to use a loop to print one element of an array at a time.

TIP

When writing a solution using pseudocode, always use a loop to print the contents of an array.

TASK 14.13

1 Write program code to implement the pseudocode from Worked Example 13.01 in Chapter 13.

2 Write program code to implement the pseudocode from Worked Example 13.02 in Chapter 13.

3 Write program code to implement the improved algorithm from Worked Example 13.03 in Chapter 13.

Creating 2D arrays

In pseudocode, a 2D array declaration is written as:

```
DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>,
<lBound2>:<uBound2>] OF <dataType>
```

Syntax definitions

Python	In Python, there are no arrays. The equivalent data structure is a list of lists.
VB.NET	`Dim <arrayIdentifier>(<uBound1, uBound2>) As <dataType>`
Java	`<dataType> <arrayIdentifier>;` `<arrayIdentifier> = new <datatype>[<uBound1>][<uBound2>];`

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

```
DECLARE Board : ARRAY[1:6, 1:7] OF INTEGER
```

Code examples

Python

```
Board = [[0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0]]
Board = [[0 for i in range(7)]
  for j in range(6)]
Board = [[0] * 7] * 6
```

2D lists can be initialised in a similar way to 1D lists.
Remember that elements are numbered from 0.
These are alternative ways of initialising a 6 × 7 list. The rows are numbered 0 to 5 and the columns 0 to 6.
The upper value of the range is not included.

VB.NET

```
Dim Board(6, 7) As Integer
```

Elements are numbered from 0 to the given number. This declaration has one row and one column too many. However, the algorithm may be such that it is easier to convert to program code if row 0 and column 0 are ignored.

Java

```
int[][] board = {
         {0, 0, 0, 0, 0, 0, 0},
```

2D arrays can be initialised in a similar way to 1D arrays.
Remember that elements are numbered from 0.

```
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0}
      }
int[][] board;
board = new int[6][7];
```

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

`<arrayIdentifier>[x, y]`

Pseudocode example:

```
Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

The following code examples demonstrate how to access elements in each of the three languages.

Code examples

Python	`Board[2][3] = 0`	Elements are numbered from 0 in Python, so [3] gives access to the fourth element.
VB.NET	`Board(3, 4) = 0`	We are ignoring row 0 and column 0.
Java	`board[2][3] = 0;`	Elements are numbered from 0 in Java, so [3] gives access to the fourth element.

TASK 14.14

1. Write program code to implement the pseudocode from Worked Example 13.04 in Chapter 13; first initialise the table and then output its contents.
2. Write program code to implement the pseudocode from Worked Example 13.05 in Chapter 13.

14.15 Text files

Writing to a text file

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE       // open the file for writing
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>                // close file
```

The following code examples demonstrate how to open, write to and close a file called `SampleFile.TXT` in each of the three languages. If the file already exists, it is overwritten as soon as the file handle is assigned by the 'open file' command.

Code examples

Python	`FileHandle = open("SampleFile.TXT", "w")`
	`FileHandle.write(LineOfText)`
	`FileHandle.close()`
		You specify the filename and mode ('w' for write) when you call the `open` function. The line of text to be written to the file must contain the newline character "\n" to move to the next line of the text file.

VB.NET	`Dim FileHandle As IO.StreamWriter`
	`Dim LineOfText As String`
	`FileHandle = New IO.StreamWriter("SampleFile.TXT")`
	`FileHandle.WriteLine(LineOfText)`
	`FileHandle.Close()`
		The file is accessed through an object (see Chapter 27) called a `StreamWriter`.

	`Alternative method:`
	`Dim LineOfText As String`
	`Dim Channel As Integer = 1`
	`FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Output, OpenAccess.Write)`
	`FileSystem.PrintLine(Channel, LineOfText)`
	`FileSystem.FileClose(Channel)`	

Java	`import java.io.FileWriter;`
	`import java.io.PrintWriter;`
	`import java.io.IOException;`
	`FileWriter fileHandle = new FileWriter("SampleFile.TXT", false);`
	`PrintWriter printLine = new PrintWriter(fileHandle);`
	`String lineOfText;`
	`printLine.printf("%s"+"%n", lineOfText);`
	`printLine.close();`
		Input output operations throw exceptions. The easiest way to manage these is to change your main heading to:
		`public static void main(String[] args) throws IOException`

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ          // open file for reading
READFILE <filename>, <stringVariable>  // read a line of text from the file
CLOSEFILE <filename>                  // close file
```

The following code examples demonstrate how to open, read from and close a file called `SampleFile.TXT` in each of the three languages.

Code examples

Python	`FileHandle = open("SampleFile.TXT", "r")`
	`LineOfText = FileHandle.readline()`
	`FileHandle.close ()`
		You specify the filename and mode ('r' for read) when you call the `open` function.

VB.NET	`Dim LineOfText As String`
	`Dim FileHandle As IO.StreamReader`
	`FileHandle = New IO.StreamReader("SampleFile.TXT")`
		The file is accessed through an object (see Chapter 27) called a `StreamReader`.

```
LineOfText = FileHandle.ReadLine()
FileHandle.Close()
```

Alternative method:

```
Dim LineOfText As String
Dim Channel As Integer = 1
FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Input, OpenAccess.Read)
FileSystem.Input(Channel, LineOfText)
FileSystem.FileClose(Channel)
```

Java

```
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;
FileReader fileHandle = new
FileReader("SampleFile.TXT");
BufferedReader textReader = new
BufferedReader(fileHandle);
String lineOfText = textReader.readLine();
textReader.close();
```

There are other library classes that can be used for input/output, such as Scanner.

Appending to a text file

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND      // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>                // close file
```

The following code examples demonstrate how to open, append to and close a file called `SampleFile.TXT` in each of the three languages.

Code examples

Python

```
FileHandle = open("SampleFile.TXT", "a")
FileHandle.write(LineOfText)
FileHandle.close()
```

You specify the filename and mode (‘a’ for append) when you call the `open` function.

VB.NET

```
Dim FileHandle As IO.StreamWriter
FileHandle = New
IO.StreamWriter("SampleFile.TXT", True)
FileHandle.WriteLine(LineOfText)
FileHandle.Close()
```

The file is accessed through a `StreamWriter`. The extra parameter, `True`, tells the system to append to the object.

Alternative method:

```
Dim LineOfText As String
Dim Channel As Integer = 1
FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Append, OpenAccess.ReadWrite)
FileSystem.Print(Channel, LineOfText)
FileSystem.FileClose(Channel)
```

Java

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
FileWriter fileHandle = new
FileWriter("SampleFile.TXT"), true);
PrintWriter printLine = new PrintWriter(fileHandle);
String lineOfText;
printLine.printf("%s"+"%n", lineOfText);
printLine.close();
```

Input output throws exceptions. The easiest way is to change your main heading to:

```
public static void main(String[] args)
throws IOException
```

The end-of-file (EOF) marker

The following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
    READFILE "Test.txt", TextString
    OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

The following code examples demonstrate how to read and then output the contents of a file in each of the three languages.

Code examples

Python

```
FileHandle = open("Test.txt", "r")
LineOfText = FileHandle.readline()
while len(LineOfText) > 0:
   LineOfText = FileHandle.readline()
   print(LineOfText)
FileHandle.close()
```

There is no explicit EOF function. However, when a line of text has been read that only consists of the end-of-file marker, the line of text is of length 0.

VB.NET

```
Dim LineOfText As String
Dim FileHandle As System.IO.StreamReader
FileHandle = New
System.IO.StreamReader("Test.txt")
Do Until FileHandle.EndOfStream
   LineOfText = FileHandle.ReadLine()
   Console.WriteLine(LineOfText)
Loop
FileHandle.Close()
```

When the end-of-file marker is detected, the `EndOfStream` method returns the value True and so the loop will end.

VB.NET

Alternative method:

```
Dim LineOfText As String
Dim Channel As Integer = 1
FileSystem.FileOpen(Channel, "Test.txt",
OpenMode.Input, OpenAccess.Read)
Do While Not FileSystem.EOF(Channel)
   FileSystem.Input(Channel, LineOfText)
   Console.WriteLine(LineOfText)
Loop
FileSystem.FileClose(Channel)
```

Java

```
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;
FileReader fileHandle = new FileReader("Test.txt");
BufferedReader textReader = new
BufferedReader(fileHandle);
String lineOfText = textReader.readLine();
while (lineOfText != null)
{
   System.out.println(lineOfText);
   lineOfText = textReader.readLine();
}
textReader.close();
```

There is no explicit EOF function. However, when a line of text has been read that only consists of the end-of-file marker, the line of text is effectively null.

TASK 14.15

Fred surveys the students at his college to find out their favourite hobby. He wants to present the data as a tally chart.

Fred plans to enter the data into the computer as he surveys the students. After data entry is complete, he wants to output the total for each hobby.

1	Reading books	\\\
2	Play computer games	\\\\\\\\
3	Sport	\\\\\
4	Programming	\\
5	Watching TV	\\\\\\\\\\\

He starts by writing an algorithm:

```
Initialise Tally array
REPEAT
```

```
    INPUT Choice // 1 for Reading, 2 for computer games,
                 // 3 for Sport, 4 for Programming, 5 for TV
                 // 0 to end input
    Increment Tally[Choice]
UNTIL Choice = 0
FOR Index = 1 TO 5
    OUTPUT Tally[Index]
NEXT Index
```

1 Write program code to declare and initialise the array `Tally : ARRAY[1:5] OF INTEGER`.

2 Write program code to implement the algorithm above.

3 Write program code to declare an array to store the hobby titles and rewrite the `FOR` loop of your program in part 2 so that the hobby title is output before each tally.

4 Write program code to save the array data in a text file.

5 Write program code to read the data from the text file back into the initialised array.

Reflection Point:

How much practice have you had writing programs? Did you get them to work?

How difficult did you find the different constructs?

Put the following in order of difficulty:

- Using a FOR loop
- Using a WHILE loop
- Using an IF ELSE statement
- Declaring a variable of a standard data type
- Declaring and using a constant
- Using a 1D array
- Using a nested loop to access each element in a 2D array
- Reading from and writing to a text file
- Using a built-in function
- Writing a procedure and calling it from the main program
- Writing a function and using its return value in an expression in the main program
- Using parameters with procedures and functions

Summary

- Programming constructs in Python, VB.NET and Java include:
  - declaration and assignment of constants and variables
  - the basic constructs of assignment, selection, repetition, input and output
  - built-in data types and functions.
- Code should be commented where it helps understanding.
- Boolean expressions are needed for conditions.
- Declaration of subroutines (functions and procedures) is done before the main program body.
- Calling a procedure is a program statement.

- Calling a function is done within an expression, for example an assignment.
- VB.NET and Java functions return exactly one value.
- Parameters can be passed to a subroutine. This is known as the interface.
- VB.NET passes parameters by value, as a default, but can return one or more values via parameters if they are declared as reference parameters.
- In Python, parameters can only pass values into a subroutine. The only way to update a value of a variable in the calling program is to return one or more values from a function.
- In Java, parameters can only pass values into a subroutine. The only way to update a value of a variable in the calling program is to return one value from a function. Note that object parameters are always passed by reference.
- When a subroutine is defined, parameters are the ‘placeholders’ for values passed into a subroutine.
- Arguments are the values passed to the subroutine when it is called.

Exam-style Questions

1 Preeti wants a program to output a conversion table for ounces to grams (1 ounce is 28.35 grams). She writes an algorithm using Structured English:

```
OUTPUT "Ounces Grams"
FOR Ounces FROM 1 TO 30
    SET Grams TO Rounded(Ounces * 28.35) // whole number of grams only
    OUTPUT Ounces and Grams
```

Write pseudocode to implement the algorithm. Include formatting, so that the output is tabulated. [7]

2 Write pseudocode to accept an input string `UserID`. The pseudocode is to test the `UserID` format. A valid format `UserID` consists of three upper case letters and four digits. The program is to output a message whether `UserID` is valid or not. [5]

3 Write pseudocode for a procedure `OutputTimesTable` that takes one integer parameter, `n`, and outputs the times table for `n`. For example the procedure call `OutputTimesTable(5)` should produce:

1 × 5 = 5
2 × 5 = 10
3 × 5 = 15
4 × 5 = 20
5 × 5 = 25
6 × 5 = 30
7 × 5 = 35
8 × 5 = 40
9 × 5 = 45
10 × 5 = 50 [6]

4 Write pseudocode for a function `isDivisible()` that takes two integer parameters, `x` and `y`. The function is to return the value True or False to indicate whether `x` is exactly divisible by `y`. For example, `isDivisible(24, 6)` should return True and `isDivisible(24, 7)` should return False. [6]

5 A poultry farm packs eggs into egg boxes. Each box takes six eggs. Boxes must not contain fewer than six eggs.

Write pseudocode for a procedure `EggsIntoBoxes` that takes an integer parameter, `NumberOfEggs`. The procedure is to calculate how many egg boxes can be filled with the given number of eggs and how many eggs will be left over. The procedure is to return two values as parameters, `NumberOfBoxes` and `EggsLeftOver`. [9]

6 In a certain country, car registrations consist 7 alphanumerical characters. The format of a car registration is either

LLNNLLL

or

LLLNNLL

where L is any capital letter and N is any numeral 0 to 9.

Use pseudocode to write a function that takes a string as parameter and returns TRUE if the format is valid and FALSE otherwise.

The string-handling functions available are those listed in Table 14.06. [7]

Chapter 15:
Software development

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the purpose of a development life cycle
- show understanding of the need of different development life cycles depending on the program being developed
- describe the principles, benefits and drawbacks of each type of life cycle: waterfall, iterative and rapid application development
- show understanding of the analysis, design, coding, testing and maintenance stages in the program development cycle
- use a structure chart to decompose a problem into sub-tasks
- use a structure chart to express the parameters passed between the various modules/procedures/functions which are part of the algorithm design
- describe the purpose of a structure chart
- construct a structure chart for a given problem
- derive equivalent pseudocode from a structure chart
- show understanding of the purpose of state-transition diagrams to document an algorithm
- show understanding of ways of exposing and avoiding faults in programs
- locate and identify the different types of errors (syntax errors, logic errors and run-time errors)
- correct identified errors
- show understanding of available testing methods (dry-run, walkthrough, white-box, black-box, integration, alpha, beta, acceptance, stub)
- select appropriate data for a given testing method
- show understanding of the need for a test strategy and test plan and their likely contents
- choose appropriate test data for a test plan (normal, abnormal, extreme/boundary)
- show understanding of the need for continuing maintenance of a system and the differences between each type of maintenance (corrective, adaptive, perfective)
- analyse an existing program and make amendments to enhance functionality.

system("COLOR
printf("Now the background
printf("Press any key to exit...");
getch();

15.01 Stages in the program development life cycle

Developing a program involves different stages. You solve a problem by designing the solution using Structured English, a flowchart and / or pseudocode (see Chapters 12 and 13). You write the program code and test it.

When large software systems are required to solve big problems, these stages are more formal, especially when more people are involved in the development. Before a solution can be designed, the problem needs to be analysed. When the program works and is being used, issues might arise that require changes. This is known as maintenance.

Analysis

The first step in solving a problem is to investigate the issues and the current system if there is one. The problem needs to be defined clearly and precisely. A ‘requirements specification’ is drawn up.

The next step is planning a solution. Sometimes there is more than one solution. You need to decide which is the most appropriate.

The third step is to decide how to solve the problem:

- bottom-up: start with a small sub-problem and then build on this
- top-down: stepwise refinement using pseudocode, flowcharts or structure charts.

Design

You have a solution in mind. How do you design the solution in detail? Chapter 12 (Section 12.05) showed that an identifier table is a good starting point. This leads you to thinking about data structures: do you need a 1D array or a 2D array to store data while it is processed? Do you need a file to store data long-term?

Plan your algorithm by drawing a flowchart or writing pseudocode.

Coding

When you have designed your solution, you might need to choose a suitable high-level programming language. If you know more than one programming language, you have to weigh up the pros and cons of each one. Looking at Chapter 14, you need to decide which programming language would best suit the problem you are trying to solve and which language you are most familiar with.

TIP

This stage is often referred to as implementation.

You implement your algorithm by converting your pseudocode into program code. When you start writing programs you might find it takes several attempts before the program compiles. When it finally does, you can execute it. It might ‘crash’, meaning that it stops working. In this case, you need to debug the code. The program might run and give you some output. This is the Eureka moment: ‘It works!!!!’. But does the program do what it was meant to do?

Testing

Only thorough testing can ensure the program really works under all circumstances (see Sections 15.06 and 15.07).

There are several different development methodologies. These include the waterfall, the iterative and the rapid application development model.

Discussion Point:

Do you think that all programs can be totally error-free?

The program development life cycle

The program development life cycle follows the defined stages of analysis, design, coding (implementation), testing and maintenance. When maintenance no longer results in a program fit for purpose, the development starts again, therefore creating a cycle (see Figure 15.01).

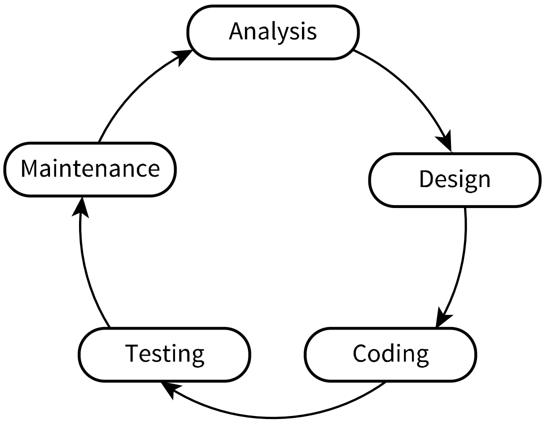


Figure 15.01 The program development life cycle

The waterfall model

Figure 15.02 shows the waterfall model.

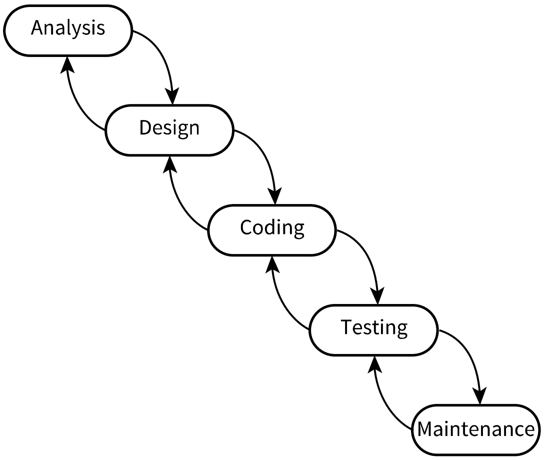


Figure 15.02 The waterfall model

The arrows going down represent the fact that the results from one stage are input into the next stage. The arrows leading back up to an earlier stage reflect the fact that often more work is required at an earlier stage to complete the current stage.

Benefits include the following.

- Simple to understand as the stages are clearly defined.
- Easy to manage due to the fixed stages in the model. Each stage has specific outcomes.
- Stages are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Drawbacks include the following.

- No working software is produced until late during the life cycle.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Cannot accommodate changing requirements.

- It is difficult to measure progress within stages.
- Integration is done at the very end, which doesn't allow identifying potential technical or business issues early.

The iterative model

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development starts with the implementation of a small subset of the program requirements. Repeated (iterative) reviews to identify further requirements eventually result in the complete system.

Benefits include the following.

- There is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development means corrective measures can be taken more quickly.
- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging of a smaller subset of program is easy.
- Risks are identified and resolved during iteration.
- Easier to manage risk – high-risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilised/applied to the next increment.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early, which facilitates customer evaluation and feedback.

Drawbacks include the following.

- Only large software development projects can benefit because it is hard to break a small software system into further small serviceable modules.
- More resources may be required.
- Design issues might arise because not all requirements are gathered at the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.

The Rapid Application Development (RAD) model

RAD is a software development methodology that uses minimal planning. Instead it uses prototyping. A prototype is a working model of part of the solution.

In the RAD model, the modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. There is no detailed preplanning. Changes are made during the development process.

The analysis, design, code and test phases are incorporated into a series of short, iterative development cycles.

Benefits include the following.

- Changing requirements can be accommodated.
- Progress can be measured.
- Productivity increases with fewer people in a short time.
- Reduces development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

Drawbacks include the following.

- Only systems that can be modularised can be built using RAD.
- Requires highly skilled developers/designers.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for projects requiring shorter development times.

15.02 Program design using structure charts

An alternative approach to modular design is to choose the sub-tasks and then construct a **structure chart** to show the interrelations between the modules. Each box of the structure chart represents a module. Each level is a refinement of the level above.

A structure chart also shows the interface between modules, the variables. These variables are referred to as ‘parameters’ (see Section 14.10). A **parameter** supplying a value to a lower-level module is shown as a downwards pointing arrow. A parameter supplying a new value to the module at the next higher level is shown as an upward pointing arrow.

Figure 15.03 shows a structure chart for a module that calculates the average of two numbers. The top-level box is the name of the module, which is refined into the three sub-tasks of Level 1. The input numbers (parameters Number1 and Number2) are passed into the ‘Calculate Average’ sub-task and then the Average parameter is passed into the ‘OUTPUT Average’ sub-task. The arrows show how the parameters are passed between the modules. This parameter passing is known as the ‘interface’.

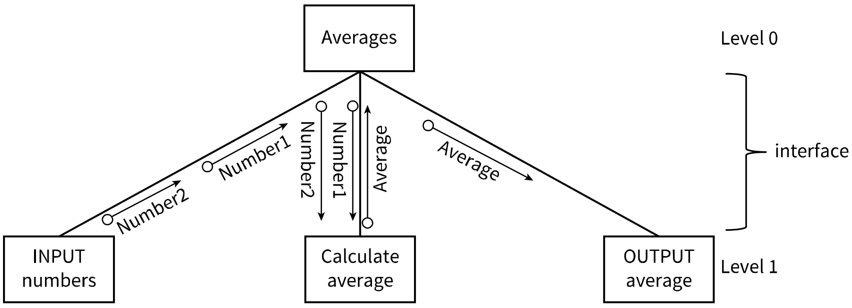


Figure 15.03 Structure chart for a module that calculates the average of two numbers

TASK 15.01

Draw a structure chart for the following module: Input a number of km, output the equivalent number of miles.

Structure charts can also show control information: selection and repetition.

The simple number-guessing game that was introduced in Chapter 12 (Section 12.06) could be modularised and presented as a structure chart, as shown in Figure 15.04.

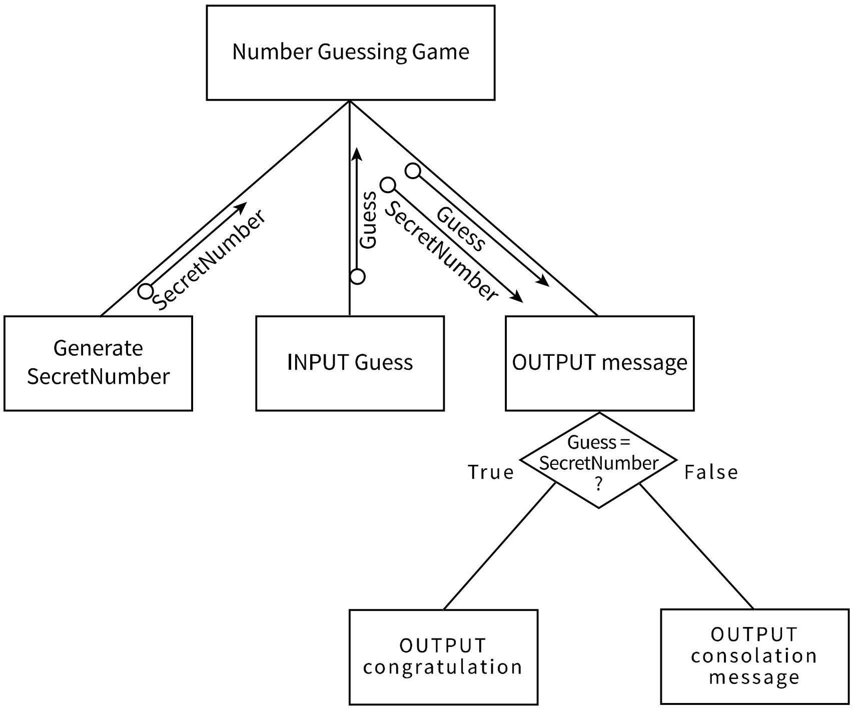

Number Guessing Game
SecretNumber
Guess
SecretNumber
Guess
Generate SecretNumber
INPUT Guess
OUTPUT message
Guess = SecretNumber ?
True
False
OUTPUT congratulation
OUTPUT consolation message

Figure 15.04 Structure chart for number-guessing game with only one guess allowed

The diamond shape shows a condition that is either True or False. So either one branch or the other will be followed.

Figure 15.05 shows the structure chart for the pyramid-drawing program from Worked Example 12.10. The semi-circular arrow represents repetition of the modules below the arrow. The label shows the condition when repetition occurs.

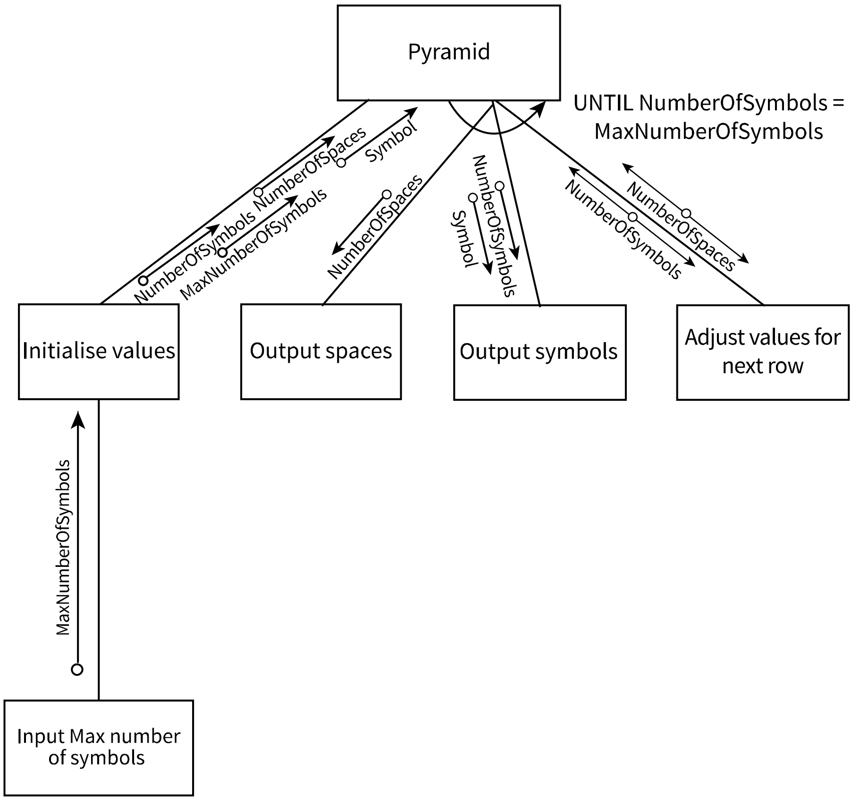

Pyramid
UNTIL NumberOfSymbols = MaxNumberOfSymbols
NumberOfSymbols
NumberOfSpaces
MaxNumberOfSymbols
Symbol
NumberOfSpaces
NumberOfSymbols
Symbol
NumberOfSymbols
NumberOfSpaces
Initialise values
Output spaces
Output symbols
Adjust values for next row
MaxNumberOfSymbols
Input Max number of symbols

Figure 15.05 Structure chart for pyramid-drawing program

TASK 15.02

Amend the structure chart for the number-guessing game (Figure 15.04) to include repeated guesses until the player guesses the secret number. The output should include the number of guesses made.

TASK 15.03

Draw a structure chart for the following problem: A user attempts to log on with a user ID. User IDs and passwords are stored in two 1D arrays (lists). The algorithm searches the list of user IDs and looks up the password in the password list. The user is given three chances to input the correct password. If the correct password is entered, a suitable message is output. If the third attempt is incorrect, a warning message is output.

Structure charts help programmers to visualise how modules are interrelated and how they interface with each other. When looking at a larger problem this becomes even more important. Figure 15.06 shows a structure chart for the Connect 4 program (Task 13.06). It uses the following symbols:

- An arrow with a solid round end ●⟶ shows that the value transferred is a flag (a Boolean value)
- A double-headed arrow ←○→ shows that the variable value is updated within the module.

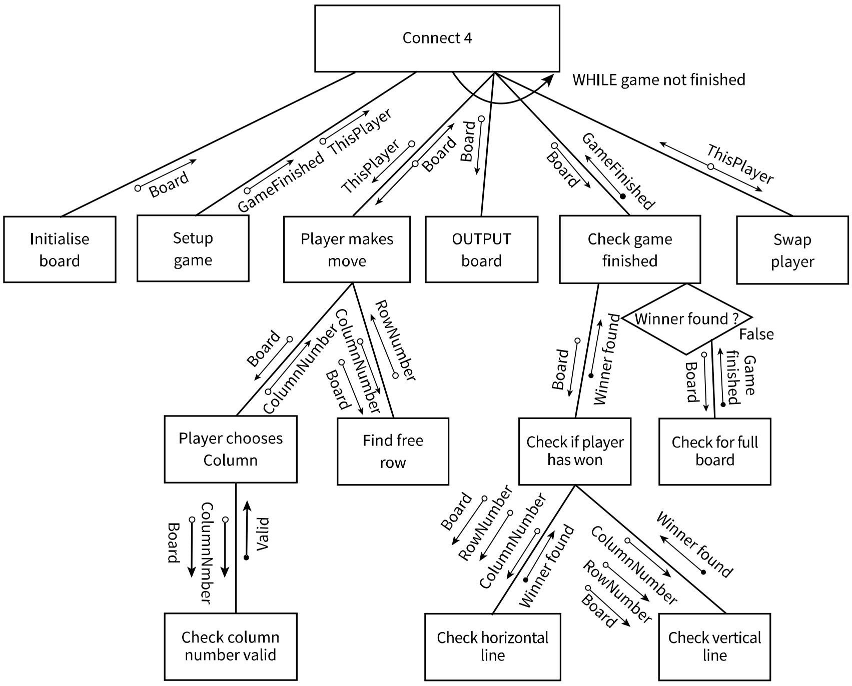

Connect 4
WHILE game not finished
Board
GameFinished
ThisPlayer
ThisPlayer
Board
Board
Board
GameFinished
ThisPlayer
Initialise board
Setup game
Player makes move
OUTPUT board
Check game finished
Swap player
Board
ColumnNumber
ColumnNumber
Board
RowNumber
Winner found ?
False
Board
Winner found
Board
Game finished
Player chooses Column
Find free row
Check if player has won
Check for full board
Board
ColumnNumber
Valid
Board
RowNumber
ColumnNumber
Winner found
ColumnNumber
RowNumber
Board
Winner found
Check column number valid
Check horizontal line
Check vertical line

Figure 15.06 Structure chart for the Connect 4 game

15.03 Deriving pseudocode from a structure chart

Let’s look at the pyramid problem again (Figure 15.05). In Worked Example 12.10, a modular solution was created without using a structure chart and all variables were global. Now we are going to use local variables and parameters. The reason for using local variables and parameters is that modules are then self-contained and any changes to variables do not have accidental effects on a variable value elsewhere.

The top-level module, Pyramid, calls four modules. When a module is called, we supply the parameters in parentheses after the module identifier. This gives the following pseudocode:

```
MODULE Pyramid
    CALL SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    REPEAT
        CALL OutputSpaces(NumberOfSpaces)
        CALL OutputSymbols(NumberOfSymbols, Symbol)
        CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    UNTIL NumberOfSymbols > MaxNumberOfSymbols
ENDMODULE

PROCEDURE SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    INPUT Symbol
    CALL InputMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE

PROCEDURE InputMaxNumberOfSymbols(MaxNumberOfSymbols)
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE

PROCEDURE OutputSpaces(NumberOfSpaces)
    FOR Count ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count
ENDPROCEDURE

PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
    FOR Count← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT Newline // move to the next line
ENDPROCEDURE

PROCEDURE AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    NumberOfSpaces ← NumberOfSpaces – 1
    NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

Note that a structure chart does not give details about how parameters are passed: by reference or by value.

TASK 15.04

1. Write pseudocode to implement the structure chart from Figure 12.03 (for the average of two numbers).
2. Write pseudocode to implement the structure chart from Figure 12.04 (for the number-guessing game).
3. Amend the pseudocode from Worked Example 13.05 to implement the interface shown in the structure chart from Figure 15.06.

Discussion Point:

The full rules of Connect 4 are that a diagonal of four tokens also is a winning line. Where in Figure

15.06 should the module to check for a diagonal be added? What parameters are required for this module? Does this additional module require further stepwise refinement?

15.04 Program design using state-transition diagrams

A computer system can be seen as a **finite state machine (FSM)**. An FSM has a start state. An input to the FSM produces a transformation from one state to another state.

The information about the states of an FSM can be presented in a **state-transition table**.

Table 15.01 shows an example FSM represented as a state-transition table.

- If the FSM is in state S1, an input of a causes no change of state.
- If the FSM is in state S1, an input of b transforms S1 to S2.
- If the FSM is in state S2, an input of b causes no change of state.
- If the FSM is in state S2, an input of a transforms S2 to S1.

A **state-transition diagram** can be used to describe the behaviour of an FSM. Figure 15.07 shows the start state as S1 (denoted by ●⟶). If the FSM has a final state (also known as the halting state), this is shown by a double-circled state (S1 in the example).

		current state	
		S1	**S2**
input	**a**	S1	S1
	b	S2	S2

Table 15.01 State-transition table

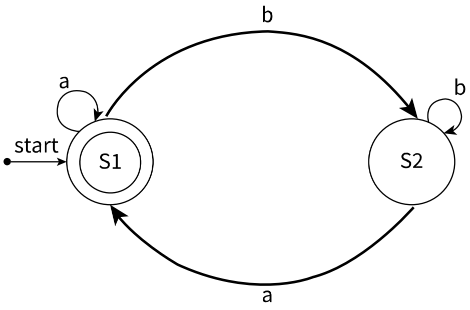


Figure 15.07 State-transition diagram

If an input causes an output this is shown by a vertical bar (as in Figure 15.08). For example, if the current state is S1, an input of b produces output c and transforms the FSM to state S2.

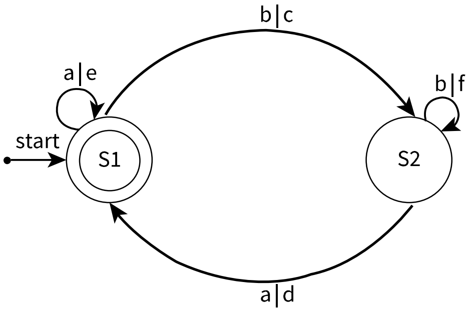


Figure 15.08 State-transition diagram with outputs

A Finite State Machine with outputs is also known as a Mealy Machine.

WORKED EXAMPLE 15.01

Creating a state-transition diagram for an intruder detection system

A program is required that simulates the behaviour of an intruder detection system.

Description of the system: The system has a battery power supply. The system is activated when the start button is pressed. Pressing the start button when the system is active has no effect. To de-activate the system, the operator must enter a PIN. The system goes into alert mode when a sensor is activated. The system will stay in alert mode for two minutes. If the system has not been de-activated within two minutes an alarm bell will ring.

We can complete a state-transition table (Table 15.02) using the information from the system description.

Current state	Event	Next state
System inactive	Press start button	System active
System active	Enter PIN	System inactive
System active	Activate sensor	Alert mode
System active	Press start button	System active
Alert mode	Enter PIN	System inactive
Alert mode	2 minutes pass	Alarm bell ringing
Alert mode	Press start button	Alert mode
Alarm bell ringing	Enter PIN	System inactive
Alarm bell ringing	Press start button	Alarm bell ringing

Table 15.02 State-transition table for intruder detection simulation

The start state is 'System inactive'. We can draw a state-transition diagram (Figure 15.09) from the information in Table 15.02.

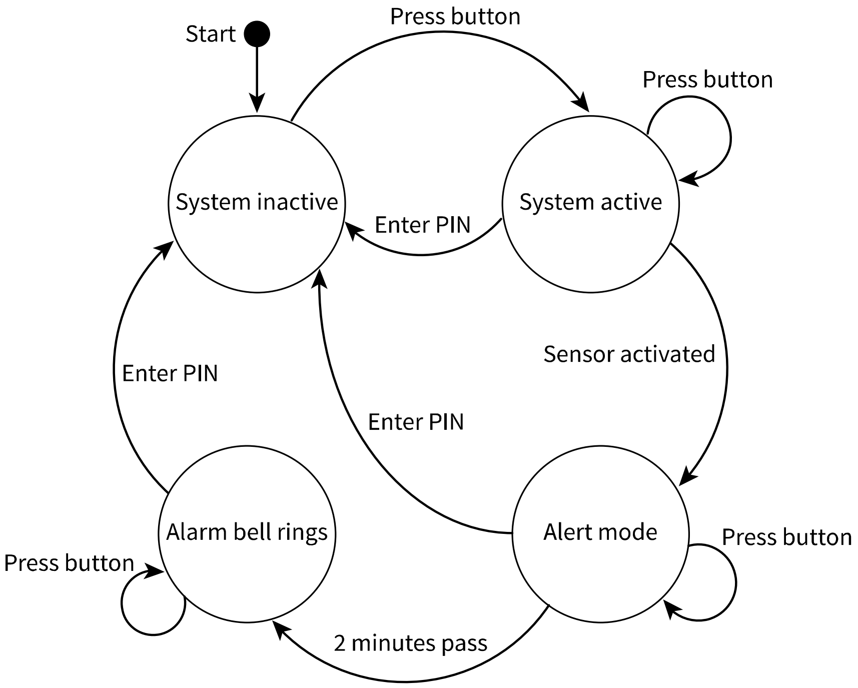

Start
Press button
Press button
System inactive
Enter PIN
System active
Sensor activated
Enter PIN
Enter PIN
Alarm bell rings
Alert mode
Press button
Press button
2 minutes pass

Figure 15.09 State-transition diagram for intruder detection system

WORKED EXAMPLE 15.02

Creating a state-transition diagram for a two's complement FSM

A finite state machine has been designed that will take as input a positive binary integer, one bit at a time, starting with the least significant bit. The FSM converts the binary integer into the two's complement negative equivalent. The method to be used is as follows.

1 Output the bits input up to and including the first 1.

2 Output the other bits following this scheme:

2.1 For each 1, output a 0.

2.2 For each 0, output a 1.

This information is represented in the state-transition table shown in Table 15.03.

Current state	S1	S1	S2	S2
Input bit	0	1	0	1
Next state	S1	S2	S2	S2
Output bit	0	1	1	0

Table 15.03 State-transition table with outputs

This method can be represented as the state-transition diagram in Figure 15.10.

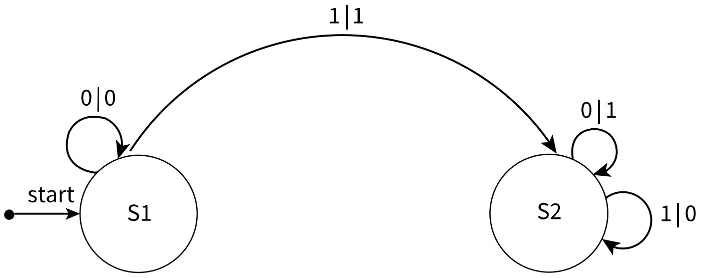


Figure 15.10 State-transition diagram for a two's complement FSM

TASK 15.05

Write a program that simulates the intruder detection system in Worked Example 15.01.

Question 15.01

What is the output from the FSM represented by the state-transition diagram in Figure 15.10, when the input is `0101`?

Extension Question 15.01

Does the FSM in Figure 15.10 work for converting a negative binary number into its positive equivalent?

15.05 Types of error

Why errors occur and how to find them

Software may not perform as expected for a number of reasons, such as:

- the programmer has made a coding mistake
- the requirement specification was not drawn up correctly
- the software designer has made a design error
- the user interface is poorly designed, and the user makes mistakes
- computer hardware experiences failure.

How are errors found? The end user might report an error. This is not good for the reputation of the software developer. Testing software before it is released for general use is essential. Research has shown that the earlier an error can be found, the cheaper it is to fix it. It is very important that software is tested throughout its development.

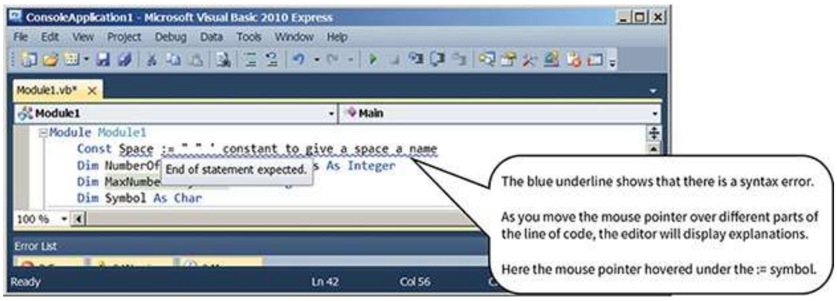
The purpose of testing is to discover errors. Edsger Dijkstra, a famous Dutch computer scientist, said ‘Program testing can be used to show the presence of bugs, but never to show their absence!’.

Finding syntax errors is easy. The compiler/interpreter will find them for you and usually gives you a hint as to what is wrong.

Depending on your development environment editor, some syntax errors may be flagged up by your editor, so you can correct these as you go along. A **syntax error** is a ‘grammatical’ error, in which a program statement does not follow the rules of the high-level language constructs.

Some syntax errors might only become apparent when you are using an interpreter or compiler to translate your program. Interpreters and compilers work differently (see Chapter 8, Section 8.05, and Chapter 20, Section 20.06). When a program compiles successfully, you know there will be no syntax errors remaining.

This is not the case with interpreted programs. Only statements that are about to be executed will be syntax checked. So, if your program has not been thoroughly tested, it might even have syntax errors remaining.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb*
Module1
Main
100 %
Error List
Ready
Ln 42
Col 56

```
Module Module1
    Const Space := " " ' constant to give a space a name
    Dim NumberOf                    s As Integer
    Dim MaxNumbe
    Dim Symbol As Char
```

End of statement expected.

The blue underline shows that there is a syntax error.

As you move the mouse pointer over different parts of the line of code, the editor will display explanations.

Here the mouse pointer hovered under the := symbol.

Figure 15.11 Syntax error in a Visual Basic program

Figure 15.11 gives an example of how a compiler flags a syntax error. The compiler stops when it first notices a syntax error. The error is often on the previous line. The compiler can’t tell until it gets to the next line of code and finds an unexpected keyword.

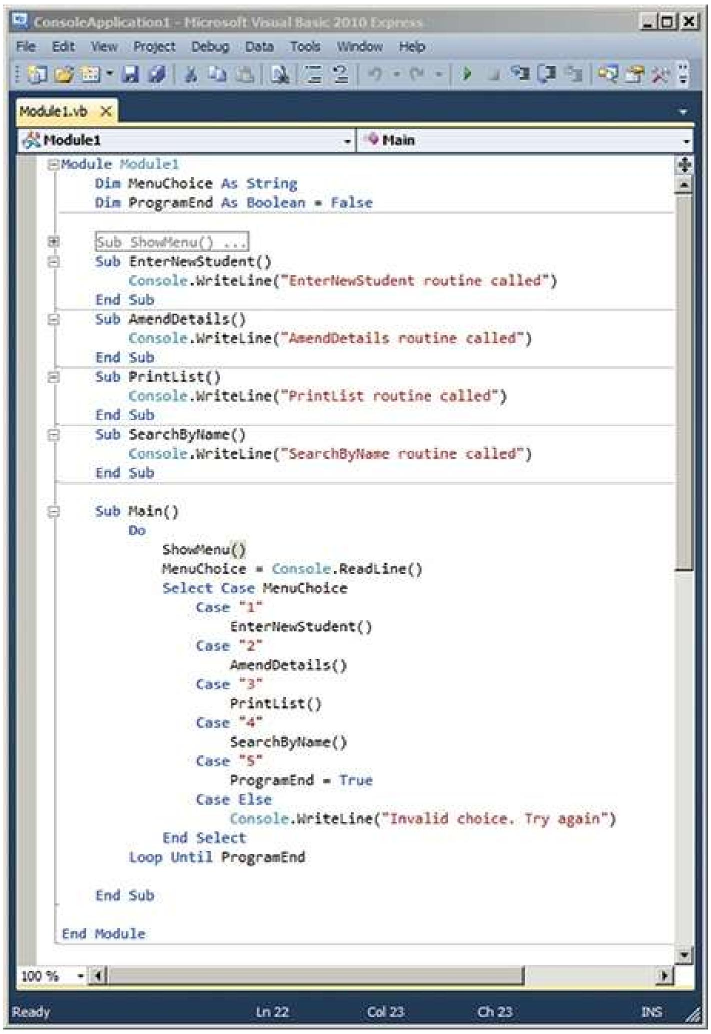
Much more difficult to find are **logic errors** and **run-time errors**. A run-time error occurs when program execution comes to an unexpected halt or ‘crash’ or it goes into an infinite loop and ‘freezes’.

Both of these types of error can only be found by careful testing. The danger of such errors is that they may only show up under certain circumstances. If a program crashes every time it is executed, it is obvious there is an error. If the program is used frequently and appears to work until a certain set of data causes a malfunction, that is much more difficult to discover without perhaps serious consequences.

15.06 Testing methods

Stub testing

When you develop a user interface, you might wish to test it before you have implemented all the facilities. You can write a ‘stub’ for each procedure (see Figure 15.12). The procedure body only contains an output statement to acknowledge that the call was made. Each option the user chooses in the main program will call the relevant procedure.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
Main
100 %
Ready
Ln 22
Col 23
Ch 23
INS

```
Module Module1
    Dim MenuChoice As String
    Dim ProgramEnd As Boolean = False

    Sub ShowMenu() ...
    Sub EnterNewStudent()
        Console.WriteLine("EnterNewStudent routine called")
    End Sub
    Sub AmendDetails()
        Console.WriteLine("AmendDetails routine called")
    End Sub
    Sub PrintList()
        Console.WriteLine("PrintList routine called")
    End Sub
    Sub SearchByName()
        Console.WriteLine("SearchByName routine called")
    End Sub

    Sub Main()
        Do
            ShowMenu()
            MenuChoice = Console.ReadLine()
            Select Case MenuChoice
                Case "1"
                    EnterNewStudent()
                Case "2"
                    AmendDetails()
                Case "3"
                    PrintList()
                Case "4"
                    SearchByName()
                Case "5"
                    ProgramEnd = True
                Case Else
                    Console.WriteLine("Invalid choice. Try again")
            End Select
        Loop Until ProgramEnd

    End Sub

End Module
```

Figure 15.12 VB.NET stub testing

Black-box testing

As the programmer, you can see your program code and your testing will involve knowledge of the code (see white-box testing).

As part of thorough testing, a program should also be tested by other people, who do not see the program code and don’t know how the solution was coded.

Such program testers will look at the program specification to see what the program is meant to do, devise **test data** and work out expected results. Test data usually consists of normal data values, extreme/boundary data values and erroneous/abnormal data values.

The tester then runs the program with the test data and records their results. This method of testing is called **black-box testing** because the tester can’t see inside the program code: the program is a ‘black box’.

Where the actual results don’t match the expected results, a problem exists. The programmer needs to find the reason for this discrepancy before correcting the program (see Section 15.08). Once black-box testing has established that there is an error, debugging software or dry-running have to be used to find the lines of code that need correcting.

White-box testing

How can we check that code works correctly? We choose suitable test data that checks every path through the code. This is called **white-box testing**.

WORKED EXAMPLE 15.03

White-box testing of pseudocode

This is the pseudocode from Worked Example 12.02 in Chapter 12:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN            // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE            // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

To test it, we need four sets of numbers with the following characteristics.

- The first number is the largest.
- The first number is larger than the second number; the third number is the largest.
- The second number is the largest.
- The second number is larger than the first number; the third number is the largest.

Note that it does not matter what exact values are chosen as test data. The important point is that the values differ in such a way that each part of the nested `IF` statement is checked. Table 15.04 lists four sets of test data and the results from them. The parts of the algorithm not entered for a particular set of data are greyed out. This makes it easier to see that each part has been checked after all four tests have been done.

Line of algorithm	**Test 1**	**Test 2**	**Test 3**	**Test 4**
`INPUT Number1`	15	12	12	8
`INPUT Number2`	12	8	15	12
`INPUT Number3`	8	15	8	15
`IF Number1 > Number2`	TRUE	TRUE	FALSE	FALSE
`THEN` `IF Number1 > Number3`	TRUE	FALSE		
`THEN`	Output 15			

`OUTPUT Number1`				

`ELSE`
`OUTPUT Number3`
`ENDIF`
		Output 15		

`ELSE` `IF Number2 > Number3`			TRUE	FALSE
`THEN` `OUTPUT Number2`			Output 15	

`ELSE`
`OUTPUT Number3`
`ENDIF`
`ENDIF`
				Output 15

Table 15.04 Testing the validity of the nested IF statement

Dry-running an algorithm

A good way of checking that an algorithm works as intended is to **dry-run** the algorithm using a **trace table** and different test data. This is also known as a **walk through**.

The idea is to write down the current contents of all variables and conditional values at each step of the algorithm.

WORKED EXAMPLE 15.04

Tracing an algorithm

Here is the algorithm of the number-guessing game:

```
SecretNumber ← 34
INPUT "Guess a number: " Guess
NumberOfGuesses ← 1
REPEAT
    IF Guess = SecretNumber
      THEN
        OUTPUT "You took ", NumberOfGuesses, " guesses”
      ELSE
        IF Guess > SecretNumber
          THEN
            INPUT "Guess a smaller number: " Guess
          ELSE
            INPUT "Guess a larger number: " Guess
        ENDIF
         NumberOfGuesses ← NumberOfGuesses + 1
    ENDIF
UNTIL Guess = SecretNumber
```

To test the algorithm, construct a trace table (Table 15.05) with one column for each variable used in the algorithm and also for the condition `Guess > SecretNumber`

Now carefully look at each step of the algorithm and record what happens. Note that we do not tend to write down values that don’t change. Here `SecretNumber` does not change after the initial assignment, so the column is left blank in subsequent rows.

SecretNumber	Guess	NumberOfGuesses	Guess > SecretNumber	Message
34	5	1	FALSE	...larger...
	55	2	TRUE	...smaller...

	30	3	FALSE	...larger...
	42	4	TRUE	...smaller...
	36	5	TRUE	...smaller...
	33	6	FALSE	...larger...
	34	7		... 7 guesses

Table 15.05 Trace table for number-guessing game

We only make an entry in a cell when an assignment occurs. Values remain in variables until they are overwritten. So a blank cell means that the value from the previous entry remains.

It is important to start filling in a new row in the trace table for each iteration (each time round the loop).

TIP

When learning to complete trace tables and to ensure you follow every line of code in the correct sequence, you can number the lines of the algorithm and add a column for the line numbers in your trace table (see Worked Example 15.05 Trace Table 15.06).

WORKED EXAMPLE 15.05

Tracing an algorithm

To test the improved algorithm of Worked Example 13.03 (bubble sort), dry-run the algorithm by completing the trace table (Table 15.06).

```
01 MaxIndex ← 7
02 n ← MaxIndex - 1
03 REPEAT
04     NoMoreSwaps ← TRUE
05     FOR j ← 1 TO n
06         IF MyList[j] > MyList[j + 1]
07           THEN
08             Temp ← MyList[j]
09             MyList[j] ← MyList[j + 1]
10             MyList[j + 1] ← Temp
11             NoMoreSwaps ← FALSE
12         ENDIF
13     NEXT j
14     n ← n – 1
15 UNTIL NoMoreSwaps = TRUE
```

Line Numbers	Max Index	n	No-MoreSwaps	j	MyList[j] > MyList[j + 1]	Temp	MyList						
							[1]	[2]	[3]	[4]	[5]	[6]	[7]
01, 02	7	6					5	34	98	7	41	19	25
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 07, 08, 09, 10				3	TRUE	98			7	98			
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				4	TRUE	98				41	98		
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				5	TRUE	98					19	98	
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				6	TRUE	98						25	98
11, 12			FALSE										
13, 14, 15		5											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 07, 08, 09, 10				2	TRUE	34		7	34				
11, 12			FALSE										
13, 05, 06, 12				3	FALSE								
13, 05, 06, 07, 08, 09, 10				4	TRUE	41				19	41		
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				5	TRUE	41					25	41	
11, 12			FALSE										
13, 14, 15		4											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 07, 08, 09, 10				3	TRUE	34			19	34			
11, 12			FALSE										
13,05, 06, 07, 08, 09, 10				4	TRUE	34				25	34		
11, 12			FALSE										
13, 14, 15		3											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 12				3	FALSE								
13, 14, 15		2											

Table 15.06 Trace table for improved bubble sort algorithm

TASK 15.06

Design a trace table for the following algorithm:

```
FUNCTION ConvertFromHex(HexString : STRING) RETURNS INTEGER
    DECLARE ValueSoFar, HexValue, HexLength, i : INTEGER
    DECLARE HexDigit : CHAR
    ValueSoFar ← 0
    HexLength ← Length(HexString)
    FOR i ← 1 TO HexLength
        HexDigit ← HexString[i]
        CASE OF HexDigit
          'A': HexValue ← 10
          'B': HexValue ← 11
          'C': HexValue ← 12
          'D': HexValue ← 13
          'E': HexValue ← 14
          'F': HexValue ← 15
          OTHERWISE HexValue ← StringToInt(HexDigit)
        ENDCASE
        ValueSoFar ← ValueSoFar * 16 + HexValue
    NEXT i
    RETURN ValueSoFar
ENDFUNCTION
```

Dry-run the function call `ConvertFromHex('A5')` by completing the trace table.

These testing methods are used early on in software development, for example when individual modules are written. Sometimes programmers themselves use these testing methods. In larger software development organisations, separate software testers will be employed.

Discussion Point:

Do you think that a program tester will find errors the programmer did not know about? You can try out the idea by letting your friends test a program that you think works perfectly.

Software often consists of many modules, sometimes written by different programmers. Each individual module might have passed all the tests, but when modules are joined together into one program, it is vital that the whole program is tested. This is known as **integration testing**. Integration testing is usually done incrementally. This means that a module at a time is added and further testing is carried out before the next module is added.

Software will be tested in-house by software testers before being released to customers. This type of testing is called **alpha testing**.

Bespoke software (written for a specific customer) will then be released to the customer. The customer will check that it meets their requirements and works as expected. This stage is referred to as **acceptance testing**. It is generally part of the hand-over process. On successful acceptance testing, the customer will sign off the software.

When software is not bespoke but produced for general sale, there is no specific customer to perform acceptance testing and sign off the software. So, after alpha testing, a version is released to a limited audience of potential users, known as ‘beta testers’. These beta testers will use the software and test it in their own environments. This early release version is called a beta version and the chosen users perform **beta testing**. During beta testing, the users will feed back to the software house any problems they have found, so that the software house can correct any reported faults.

15.07 Test strategy, test plans and test data

During the design stage of a software project, a suitable testing strategy must be worked out to ensure rigorous testing of the software from the very beginning. Consideration should be given to which testing methods are appropriate for the project in question. A carefully designed test plan has to be produced.

It is important to recognise that large programs cannot be exhaustively tested but it is important that systematic testing finds as many errors as possible. We therefore need a test plan. In the first instance, an outline plan is designed, for example:

- flow of control: does the user get appropriate choices and does the chosen option go to the correct module?
- validation of input: has all data been entered into the system correctly?
- do loops and decisions perform correctly?
- is data saved into the correct files?
- does the system produce the correct results?

This outline test plan needs to be made into a detailed test plan.

How can we carry out these tests? We need to select data that will allow us to see whether it is handled correctly. This type of data is called 'test data'. It differs from real, live data because it is specifically chosen with a view of testing different possibilities. We distinguish between different types of test data, listed in Table 15.07.

Type of test data	Explanation
Normal (valid)	Typical data values that are valid
Abnormal (erroneous)	Data values that the system should not accept
Boundary (extreme)	Data values that are at a boundary or an extreme end of the range of normal data; test data should include values just within the boundary (that is, valid data) and just outside the boundary (that is, invalid data)

Table 15.07 Types of test data

WORKED EXAMPLE 15.06

Designing test data

Look at the Pyramid Problem (code shown in Section 14.13). This is a simple program, but we can use it to illustrate how to choose test data. There are just two user inputs: the number of symbols that make up the base and the symbol that is to be used to construct the pyramid. Let's consider just the test data for the number of symbols (Table 15.08).

Type of test data	Example test values	Explanation
Normal (valid)	7	7 is an odd integer, so should be accepted. Any odd positive integer would be suitable as test data. However, it should be bigger than 1 to check that the pyramid is correctly formed. More than one different value to test would be a good idea.
Abnormal (erroneous)		Any number that is not a positive odd integer. This will require several tests to ensure that the following types of data are not accepted:

	-7	• negative integer
	8	• even integer
	7.5	• real number
	'*'	• non-numeric input.
		You should not take shortcuts and choose one negative even integer or one negative real number and think you can test two things at the same time. You will not know whether the test fails for just one reason or both.
Boundary (extreme)	1	What is a boundary value? The smallest possible pyramid is a single symbol. So the value 1 is just within the boundary. Sometimes choosing test data throws up some interesting questions that need to be considered when designing the solution:
	0	• Should 0 be accepted? Is 0 an even number? Is it outside the boundary because a pyramid of 0 symbols is not really a pyramid? • Is there just one boundary? Should the program reject numbers that are too large?
	79 81	The output would not look like a pyramid if there were a wrap-around. So the program really should check how many symbols fit onto one line and not allow the user to input a number greater than this. If the number of characters across the screen is 80, then 79 would be just within the boundary but 81 would be outside the boundary, and should not be accepted. Note that by testing with values within the boundary you are also testing normal data, albeit at the extreme ends of the normal range.

Table 15.08 Test data for the pyramid problem

TASK 15.07

Look at the programs you wrote in Chapter 14.

1. Design test data for the number-guessing game (Task 14.09.2).
2. Design test data for the Connect 4 game (Task 14.11).

How to prevent errors

The best way to write a program that works correctly is to prevent errors in the first place. How can we minimise the errors in a program? A major cause of errors is poor requirements analysis. When designing a solution it is very important that we understand the problem and what the user of the system wants or needs. We should use:

- tried and tested design techniques such as structured programming or object-oriented design
- conventions such as identifier tables, data structures and standard algorithms
- tried and tested modules or objects from program libraries.

15.08 Corrective maintenance

Maintaining programs is not like maintaining a mechanical device. It doesn’t need lubricating and parts don’t wear out. **Corrective maintenance** of a program refers to the work required when a program is not working correctly due to a logic error or because of a run-time error. Sometimes program errors don’t become apparent for a long time because it is only under very rare circumstances that there is an unexpected result or the program crashes. These circumstances might arise because part of the program is not used often or because the data on an occasion includes extreme values. Earlier corrective maintenance may also introduce other errors.

When a problem is reported, the programmer needs to find out what is causing the bug. To find a bug, a programmer either uses program debugging software or a trace table (see Section 15.06).

TASK 15.08

1 Design a trace table for the following algorithm:

```
INPUT BinaryString
StringLength ← Length(BinaryString)
FOR i ← 1 TO StringLength
    Bit ← BinaryString[i]
    BitValue ← IntegerValue(Bit) // convert string to integer
    DenaryValue ← DenaryValue + 2 + BitValue
NEXT i
```

2 Dry-run the algorithm using '101' as the input. Complete the trace table.

3 The result should be 5. Can you find the error in the code and correct it?

15.09 Adaptive maintenance

Programs often get changed to make them perform functions they were not originally designed to do.

For example, the Connect 4 game introduced in Chapter 13 (Worked Example 13.03) allows two players, O and X, to play against each other. An amended version would be for one player to be the computer. This would mean a single player could try and win against the computer.

Adaptive maintenance is the action of making amendments to a program to enhance functionality or in response to specification changes.

TASK 15.09

Design the algorithm to simulate the computer playing the part of Player X in Connect 4.

15.10 Perfective maintenance

The program runs satisfactorily. However, there is still room for improvement. For example, the program may run faster if the file handling is changed from sequential access to direct access.

TASK 15.10

Analyse the pseudocode below and make amendments to enhance maintainability.

```
FUNCTION GetPositiveNumber
    DECLARE n : INTEGER
    OUTPUT "Enter a positive number: "
    INPUT n
    RETURN n
ENDFUNCTION
// main program
REPEAT
    Number1 ← GetPositiveNumber
    IF Number1 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number1 > 0
REPEAT
    Number2 ← GetPositiveNumber
    IF Number2 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number2 > 0
```

Reflection Point:

Have you used dry-running for programs you have written? You can check your trace table if you add output statements at key points in your program. You can then compare the program output with the contents of your trace table.

Summary

- The stages of the program development cycle consist of analysis, design, coding, testing and maintenance.
- Structure charts are graphical representations of the modular structure of solutions.
- A structure chart shows the interface between modules: parameters passed between calling module and the module being called.
- Structure charts show selection, where a module is called only under certain conditions.
- Structure charts show repetition, where modules are called repeatedly.
- A state transition diagram is another way of documenting an algorithm.
- Testing strategies include stub testing, black-box testing, white-box testing, integration testing, alpha and beta testing, and acceptance testing.
- Locating and correcting logic errors and run-time errors can be done by dry-running an algorithm or using a trace table.
- Corrective maintenance means fixing bugs that have come to light during use of the program.
- Adaptive maintenance involves altering an algorithm and data structure in response to required

changes.

- **Perfective maintenance** means enhancing performance or maintainability.

Exam-style Questions

1 Consider this code for a function:

```
FUNCTION Binary(Number : INTEGER) RETURNS STRING
    DECLARE BinaryString : STRING
    DECLARE PlaceValue : INTEGER
    BinaryString ← ''  // empty string
    PlaceValue ← 8
    REPEAT
        IF Number >= PlaceValue
          THEN
            BinaryString ← BinaryString & '1' // concatenates two strings
            Number ← Number – PlaceValue
          ELSE
            BinaryString ← BinaryString & '0'
        ENDIF
        PlaceValue ← PlaceValue DIV 2
    UNTIL Number = 0
    RETURN BinaryString
ENDFUNCTION
```

a Dry-run the function call `Binary(11)` by completing the given trace table.

`Number`	`BinaryString`	`PlaceValue`	`Number >= PlaceValue`
11	''	8	

What is the return value? [5]

b **i** Now dry-run the function call `Binary(10)` by completing the given trace table.

`Number`	`BinaryString`	`PlaceValue`	`Number >= PlaceValue`
10	''	8	

What is the return value? [3]

ii The algorithm is supposed to convert a denary integer into the equivalent binary number, stored as a string of 0s and 1s. Explain the result of each dry-run and what needs changing in the given algorithm. [3]

2 A procedure to output a row in a tally chart has been written using pseudocode:

```
PROCEDURE OutputTallyRow(NumberToDraw : INTEGER)
    IF Count > 0
      THEN
        FOR Count ← 1 TO NumberToDraw
            IF (Count MOD 5) = 0
              THEN
                OUTPUT('\') // every 5th bar slants the other way
              ELSE
```

```
                OUTPUT('/')
            ENDIF
        NEXT Count
    ENDIF
    OUTPUT NewLine // move to next row
ENDPROCEDURE
```

Suggest suitable test data that will test the procedure adequately. Justify your choices in each case. [9]

3 A random number generator is to be tested to see whether all numbers within the range 1 to 20 are generated equally frequently. The structured English version of the algorithm is

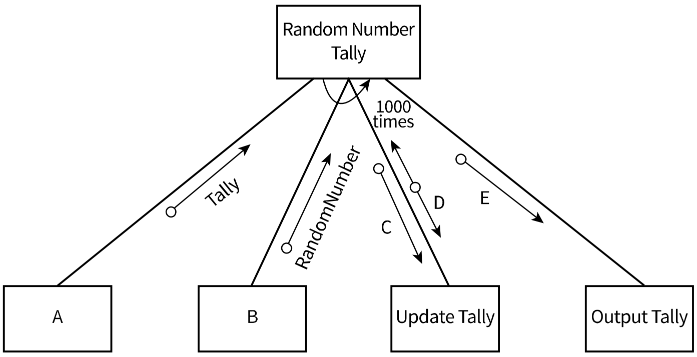
```
Initialise a tally for the numbers 1 to 20
Repeatedly generate numbers in range 1 to 20
For each number generated, increment the relevant count
Calculate how often each number should be generated (expected frequency)
Output expected frequency
Output the list of numbers as a table with actual frequency
```

The identifiers required are:

Identifier	**Data Type**	**Explanation**
`Tally`	`Array[1 : 20] OF INTEGER`	1D array to store the count of how many times each number has been generated
`RandomNumber`	`INTEGER`	The random number generated
`NumberOfTests`	`INTEGER`	The number of times a random number is to be generated (1000 in this example)
`ExpectedFrequency`	`INTEGER`	The number of times any one number would be generated if all numbers are generated equally frequently (1000/20 in this example)

a Complete the structure chart below by naming the labels A to E. [5]

b Develop pseudocode from the structure chart. [12]

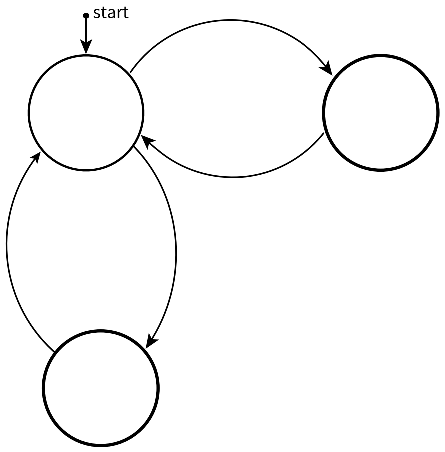


4 A car park has a barrier at the exit. The starting position of the barrier is lowered. When a car wants to exit the car park, the driver has to insert a coin into a coin slot at the barrier. The barrier raises and allows the car to drive out of the car park. After the car has passed through the barrier, the barrier lowers. In case of emergency, a member of staff can open the barrier using a remote control. The barrier will remain open until the remote control is used again to lower the barrier.

The barrier has three states: lowered, raised and open. The transition from one state to another is as shown in the state-transition table:

Current state	**Event**	**Next state**
Barrier lowered	Coin inserted	Barrier raised
Barrier lowered	Open remotely	Barrier open
Barrier open	Close remotely	Barrier lowered
Barrier raised	Car has exited	Barrier lowered

Complete the state-transition diagram for the barrier:



[7]

Part 3

Advanced theory